41.-E501 362

Copy 68 of 145 copies

(2)

IDA PAPER P-2378

AN APPROACH FOR CONSTRUCTING REUSABLE SOFTWARE COMPONENTS IN ADA

Stephen Edwards

September 1990

Prepared for
Strategic Defense Initiative Organization (SDIO)

IDA

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

IDA Log No. HQ 90-035362

The work reported in this document was conducted under contract MDA 903 89 C 0003 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE September 1990	3. REPORT TYPE AND DATES COVERED Final

4. TITLE AND SUBTITLE An Approach for Constructing Reusable Software Components in Ada	5. FUNDING NUMBERS MDA 903 89 C 0003 T-R2-597.2
6. AUTHOR(S) Stephen Edwards	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Institute for Defense Analyses (IDA) 1801 N. Beauregard Street Alexandria, VA 22311-1772	8. PERFORMING ORGANIZATION REPORT NUMBER IDA Paper P-2378
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Strategic Defense Initiative Organization (SDIO) SDIO/ENA The Pentagon, Room 3E149 Washington, DC 20301-7100	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Public release/unlimited distribution.	12b. DISTRIBUTION CODE 2A

13. ABSTRACT (Maximum 200 words)

This paper discusses the topic of software reuse and is aimed at the software engineer who may actually be designing reusable software. The paper concentrates on many of the technical problems encountered when constructing reusable software components today. This paper does not, however, focus on the general problem of reusable software design. Instead, it focuses on the Ada programming language, and the problems software engineers may encounter when designing components in this language. This paper is intended to be a companion to IDA Paper P-2494, Strategy and Mechanisms for Encouraging Reu.^e in the Acquisition of SDI Software. P-2494 discusses the managerial and legal issues involved with soft ,are reuse.

14. SUBJECT TERMS Software Reuse; Ada Programming Language; Reusable Software Components; 3C Model; Parameterization Management; Table-Driven Programming.			15. NUMBER OF PAGES 214
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT SAR

IDA PAPER P-2378

AN APPROACH FOR CONSTRUCTING REUSABLE SOFTWARE COMPONENTS IN ADA

Stephen Edwards

September 1990

INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 89 C 0003
Task T-R2-597.2

Preface

IDA Paper P-2378, *An Approach for Constructing Reusable Software Components in Ada*, was prepared for the Strategic Defense Initiative Organization (SDIO) in response to tasking contained in IDA Task Order T-R2-597.2 under contract MDA 903-89-C-0003.

This paper is intended to be a companion to IDA Paper P-2494, *Strategy and Mechanisms for Encouraging Reuse in the Acquisition of SDI Software*. P-2494 discusses the managerial and legal issues involved with software reuse, while this paper discusses the technical details of expressing reusable software in the Ada programming language. A detailed description of the paper's content is provided in the Executive Summary.

This document was reviewed internally by Richard Wexelblat, Terry Mayfield, James Baldo, Dennis Fife, Norman Howes and David Wheeler. Thanks to Will Tracz, Bruce Weide, and Larry Latour for participating in the external review process and to Sylvia Reynolds for her editorial advice and assistance.

EXECUTIVE SUMMARY

Recently, both software engineers and the managers of software projects have become increasingly more interested in the topic of software reuse. After a decade of research into software reuse techniques, members of the software industry are seriously considering which techniques should be incorporated into production projects. Of course, many approaches to software reuse are still under research, and are not yet mature enough for production use.

This paper, on the other hand, is aimed at the professional software engineer who may actually be designing reusable software. It concentrates on many of the technical problems encountered when constructing reusable software components today. This paper does not, however, focus on the general problem of reusable software design. Instead, it focuses on the Ada programming language, and the problems software engineers may encounter when designing components in this language.

This summary briefly describes the content of the remainder of this paper. It presents a distillation of the main points made in the paper, some of which are not specific to the Ada language and address software reuse in general. It also summarizes the technical contributions the paper makes to reuse research, and concludes with a brief description of the significant technical points made in the main body of the paper.

CENTRAL POINTS OF THIS WORK

The central theme upon which this work is built is a model of what reusable components are, and what this structure shows about the process of software reuse. This model, the "3C" model presented in Section 2, was originally conceived at the "Reuse in Practice" workshop held at the Software Engineering Institute in June, 1989 [Tracz90a].

Because this model is still developing, this paper actually encompasses work in progress. There is still little experimental evidence to support quantitative claims about the effectiveness of this particular reuse approach. However, the information about the technical limitations of the Ada language presented here is certain to provide new insights for software engineers new to reuse. The goal of this paper is to provide such engineers with a useful way to think about the process of reusing software, and the process of designing reusable components. It is also the goal of this paper to provide software engineers with a description of some of the problems they may encounter when designing components in the Ada language, with the hope that this will allow them to construct software that is more reusable.

Upon the foundation of the 3C model, this paper develops a set of guidelines for representing reusable components in Ada. These guidelines are designed to provoke component

designers into thinking about design decisions they might other make without taking reusability into consideration.

In addition to the central theme of the 3C model, Section 2 states that reduced maintenance cost is the true goal of software reuse. While many past researchers have concentrated on the savings reuse can offer for development, that benefit is far outweighed by the potential maintenance savings.

This paper also differs from many previous writings about software reuse in Ada. It focuses on the limitations of Ada that inhibit or reduce reusability, since Ada's benefits have been effectively discussed elsewhere. This is a direct result of the goal to provide software engineers with practical information about constructing reusable software in Ada.

Both Ada's limitations and the structure of the 3C model point out a significant problem that has not yet been solved in any language. Termed the "parameterization management" problem, it may limit the size or the flexibility of large reusable components. This problem is introduced in Section 2.6 and expanded on in Section 5.3.

In addition to focusing on Ada's limitations, this paper also discusses several areas where unwary component designers are likely to make mistakes. Often, the reuse implications of certain decisions made during design are not immediately apparent. With the goal of forewarning designers, Section 6 summarizes the most common areas where choices might be implicitly made that limit component reusability.

TECHNICAL CONTRIBUTIONS

There are five main technical contributions in this paper. First, the description of the 3C model presented in Section 2 is a significant step in the maturation of this model. Although the validity of the model is still open to debate, this presentation of the model is a step towards solidifying its concepts.

Second, the technical guidelines on how to represent reusable components in Ada is an important contribution. These guidelines are founded on the 3C model, and form the heart of the paper. The guidelines presented in this paper are not designed like a "style guide" for programmers, but rather as thought provoking statements about how the interfaces of reusable components should be designed. The purpose of these guidelines is to spur component designers and writers to think about the ramifications of decisions they might otherwise make without serious thought.

Third, the discussion on table-driven programming presented in Section 5.2.3 is also notable. Although the table-driven approach to programming is certainly not new, documentation on it is scarce. In addition, Section 5.2.3 shows how table-driven programs can be efficiently implemented in Ada while still maintaining an effective table-driven abstraction.

Fourth, the discussion of iterators presented in Section 6.3 consolidates some previous work on the topic. It also offers a somewhat new perspective on the various techniques for defining iterators, and presents a summary of the advantages and disadvantages of each approach.

Fifth, Appendix A contains a preliminary component labeling strategy. The contents of Appendix A are derived from Section 6, which is in turn derived from Booch's work [Booch87a]. Although Appendix A is primarily an aggregation of previous work, new information has been added so that it can be effectively used not only to label new components but also to effectively determine the "usability" of labeled components in a library.

TECHNICAL SUMMARY

After the background material, the technical body of this paper is divided into five parts. Section 2 presents the 3C model. Then Section 3 describes the reusable component described as an example through the paper. Section 4 briefly discusses the features of Ada that enhance reusability, and presents guidelines for using these features. Section 5 covers the limitations of Ada that restrict reusability, and Section 6 covers the areas where common mistakes that limit reusability are likely.

In Section 5, five main limitations are discussed. First, the restrictions of Ada's encapsulation mechanisms are presented, along with techniques to avoid these restrictions. Second, the fact that Ada does not support multiple implementations for a single package is discussed, along with work-arounds. Third, the fact that the Ada language is built around the assumption that the assignment, or copy, operator is the primary means of moving data around is discussed, including viable alternatives. Fourth, Ada's lack of support for table-driven programming is presented, including an alternative. Finally, the parameterization management problem is discussed.

In Section 6, four areas where common mistakes are likely are delineated. These areas consist of memory management models, concurrency protection models, iterators for abstract data types, and save and restore operations for abstract data types. The possible approaches in each area are presented, with the discussion focusing on the choices that restrict the reusability of the resulting components.

TABLE OF CONTENTS

PREFACE iii

EXECUTIVE SUMMARY v
CENTRAL POINTS OF THIS WORK v
TECHNICAL CONTRIBUTIONS vi
TECHNICAL SUMMARY vii

1. INTRODUCTION 1
1.1 PURPOSE 1
1.2 SCOPE 2

2. A MODEL OF A REUSABLE SOFTWARE COMPONENT 3
2.1 THE 3C MODEL 3
2.2 THE PROCESS OF SOFTWARE REUSE AND ITS BENEFITS 10
2.3 HOW COMPONENT TAILORING AFFECTS THE BENEFITS OF REUSE 12
2.4 POSSIBLE MISCONCEPTIONS ABOUT THE 3C MODEL 15
2.5 REUSE OF OTHER SOFTWARE ARTIFACTS 16
2.6 REUSE OF MORE COMPLEX SUBSYSTEMS 17

3. AN EXAMPLE OF A REUSABLE COMPONENT 19
3.1 DESCRIPTION OF THE GPD ABSTRACTION 20
3.2 THE ADA SPECIFICATION FOR THE GPD PACKAGE 21
3.2.1 THE GENERIC PARAMETERS AND TYPE DEFINITION IN THE GPD *CONCEPT* 21
3.2.2 THE COMMON OPERATIONS SUPPORTED FOR ALL GPD NODES 22
3.2.3 AN EXAMPLE OF THE OPERATIONS DEFINED FOR LEAF NODES 22
3.2.4 THE OPERATIONS SUPPORTED FOR *GPD_PARENT* NODES 24
3.2.5 THE OPERATIONS SUPPORTED FOR GPD_SEQUENCE NODES 25
3.2.6 THE REMAINDER OF THE GPD *CONCEPT* 26

4. BENEFITS OF ADA FOR REUSABLE SOFTWARE 29
4.1 SUPPORT FOR *CONCEPT*—PACKAGES AND LIMITED PRIVATE TYPES 29

4.2 SUPPORT FOR *CONTENT*—PACKAGE BODIES AND MULTIPLE IMPLEMENTATIONS 31
4.3 SUPPORT FOR *CONTEXT*—GENERICS AND WITH CLAUSES 32

5. LIMITATIONS OF ADA THAT RESTRICT REUSABILITY 36
5.1 SUPPORT FOR *CONCEPT*—DATA ABSTRACTION AND ENCAPSULATION 36
5.2 SUPPORT FOR *CONTENT* 43
5.2.1 Multiple Implementations 43
5.2.2 Data Movement 46
5.2.2.1 Optimizing Copy Operations 48
5.2.2.2 Using Structural Sharing Instead of Copying 49
5.2.2.3 Eliminating Copy Operations With Swapping 50
5.2.2.4 Guidelines on Data Movement 52
5.2.3 Table-Driven Programming 54
5.3 SUPPORT FOR *CONTEXT*—PARAMETERIZATION MANAGEMENT 62

6. AREAS WHERE DESIGN DECISIONS COMMONLY RESTRICT REUSABILITY 65
6.1 MEMORY MANAGEMENT 66
6.2 CONCURRENCY 68
6.3 ITERATORS 70
6.3.1 Eliminating the Need for an Explicit Iterator 71
6.3.2 Passive Iterators 73
6.3.3 Active Iterators 75
6.3.4 Recommendations on Iterator Construction 79
6.4 SAVE/RESTORE BEHAVIOR 84

7. CONCLUSIONS 93

APPENDIX A – ATTRIBUTES FOR LABELING THE INTERFACE CHARACTERISTICS OF REUSABLE ADA COMPONENTS 95

1. IMPLEMENTATION CHARACTERISTICS 97

2. CHARACTERISTICS OF GENERIC PARAMETERS TO THE COMPONENT 101

3. CHARACTERISTICS OF EXPORTED TYPES AND OPERATIONS 105

4. USING THE ATTRIBUTE VALUES 109

APPENDIX B – SUMMARY OF GUIDELINES 113

APPENDIX C – SUMMARY OF ADA UNITS USED IN EXAMPLES 121

1. "NAIVE" ADA SPECIFICATION FOR GPD *CONCEPT* (SECTION 3) 122
2. GPD *CONCEPT* SPECIFICATION, AS SUGGESTED IN SECTION 5.1 127
3. GPD *CONCEPT* SPECIFICATION, AS SUGGESTED IN SECTION 5.2.2 133
4. GPD *CONCEPT* SPECIFICATION, AS SUGGESTED IN SECTION 5.2.3 139
5. GPD *CONCEPT* SPECIFICATION, AS SUGGESTED IN SECTION 6.4 147
6. PROCEDURE VARIABLE *CONCEPT* 159
7. PROCEDURE VARIABLE *CONTENT* USING TASKING 161
8. PROCEDURE VARIABLE *CONTENT* USING *INTERFACE* PRAGMA 165
9. C PROCEDURE VARIABLE INVOCATION USED BY *PROCEDURE_VARIABLE_ABSTRACTION* PACKAGE 169
10. PROCEDURE VARIABLE *CONCEPT* WITH SHARED "ENVIRONMENTS" 171
11. UNIDIRECTIONAL ASSOCIATIVE MEMORY *CONCEPT* FROM SECTION 6.3.1 173

BIBLIOGRAPHY 175

LIST OF FIGURES

Figure 1. The 3C View of a Reusable Component 4

Figure 2. Simple "Stack" *Concept* 7

Figure 3. *Concept* with Hard-Wired *Context* 7

Figure 4. A System of *Concepts* and Implementations 8

Figure 5. Mapping the 3 "C"s into Ada 9

Figure 6. Graphical Depiction of a GPD Structure 21

Figure 7. The Generic Parameters and Type Definition Used in the Specification for GPD *Concept* 22

Figure 8. The Common Operations Exported for GPD Nodes 23

Figure 9. The Subpackage Defining the Operations for *GPD_Integer* Nodes 23

Figure 10. The Subpackage Defining the Operations for *GPD_Parent* Nodes 24

Figure 11. The Subpackage Defining the Operations for *GPD_Sequence* Nodes 26

Figure 12. The Private Part of the GPD Specification 26

Figure 13. Horizontal Coupling 33

Figure 14. Vertical Coupling 35

Figure 15. Declaration of *Common_Node_Contents* 37

Figure 16. *Common_Node_Contents* as a **Limited Private** Type 38

Figure 17. *Common_Node_Contents* with Initialization and Finalization 39

Figure 18. Declaration of *GPD_type* 39

Figure 19. **Limited Private** Declaration of *GPD_type* 40

Figure 20. *GPD_type* with Initialization and Finalization 40

Figure 21. Minimum Operations for Generic Formal Type Parameters 43

Figure 22. *GPD_type* declared with *Swap* in Addition to *Assign* 51

Figure 23. New operations for *GPD_type* 55

Figure 24. Generic for a User-Defined GPD Node 56

Figure 25. Complete Set of Generic Parameters for the *User_Defined_Node_Pkg* . . . 57

Figure 26. Procedure Variable *Concept* 59

Figure 27. Buhr Diagram of a Tasking Implementation of Procedure Variables 59

Figure 28. Modified *Procedure_Definer* which allows Shared "Environments" 61

Figure 29. Declaration of a GPD Iterator 70

Figure 30. Unidirectional Associative Memory *Concept*[18] 73

Figure 31. An Additional Primary Operation for the Unidirectional Associative Memory *Concept* 74

Figure 32. An Attempt at a Two Complete Passive Iterators 75

Figure 33. A Simple Active Iterator 76

Figure 34. An Active, Random Access Iterator 77

Figure 35. A Generic Sort Routine, Which Assumes the Existence of a Standardized Active Iterator for Sequences 80

Figure 36. Suggested Standard Profile for an Active, Sequential Iterator 82

Figure 37. Suggested Standard Profile for an Active, Random Access Iterator 83

Figure 38. A GPD Structure That Refers to Structurally Shared Data 86

Figure 39. High-Level Pseudocode for the Save Operation 87

Figure 40. Erroneously Reproduced Data Structure 88

Figure 41. Modifications to the Generic Parameters of *GPD_pkg* to Support Two-Pass Saving 90

Figure 42. Recommended Changes to the Generic Parameters of the *GPD_pkg* to Support Save and Restore Behavior 92

Figure 43. Recommended Changes to the Operations Exported by the *GPD_pkg* to Support Save and Restore Behavior 92

LIST OF TABLES

Table 1. A Comparison of Iterator Techniques 79

1. INTRODUCTION

1.1 PURPOSE

Many exciting claims have made by individuals advocating ''software reuse'': it will save money, increase productivity, reduce errors, and improve maintenance. But what is ''reusable software,'' and how are reusable software components constructed? Different individuals often have widely disparate ideas of what constitutes such software, or of what the process of ''software reuse'' is all about. To evaluate these claims, one must gain a clearer understanding of the concept of reuse and its implications. This paper is aimed at developing such an understanding.

This paper explores the topic of software reuse at a programming level, focusing on how reuse can be achieved in the Ada programming language. It is aimed at people working at the detailed design level, designing units that will be implemented in Ada and that may have reuse potential. The paper primarily discusses code reuse, but in order to achieve effective code reuse, reuse must be considered during detailed design. In fact, the earlier software reuse is considered during the design process, the greater the potential for savings. Thus, the paper presents a detailed discussion of the tradeoffs a designer must consider early on during the design of a component. This analysis is centered around a conceptual model of what a reusable component actually is. This model is a very appropriate way of thinking about reusable components during all stages of development, whether the goal is designing new components or applying available ones.

Because of the language-specific detail present through much of the paper, the reader should be very familiar with the Ada language. Although the Sections 1 through 2 can be read by a wider audience, the remainder of the paper assumes the reader is already familiar with how the features of Ada are used to create software modules. In particular, the reader should be very fluent in the use of packages to control visibility, the use of **limited private** types to model abstract data types, and the use of **generic** packages to provide configurable software modules. Readers who wish to get the most value from Sections 3 through 6, but who do not have this background knowledge, should consult references such as [Booch87a], [Tracz89a], and [Mendal86a].

1.2 SCOPE

This paper is intended to highlight the necessary software reuse design decisions that must be made during the detailed design of component interfaces, and to provide advice on how to make these decisions. Since reuse of ''software artifacts'' at a higher level than code (i.e., reuse of design, specification, etc.) is still relatively immature, this paper avoids detailed discussion of that. Instead, it focuses on code reuse techniques, though some sections herein may also be applicable to the reuse of other software products (Section 2.5).

The guidelines contained in this paper are general guidelines applicable to any Ada development effort, and not targeted to any specific real-time or SDI-related efforts. In particular, this paper raises a collection of tradeoff decisions that must be made and discusses the factors that are important to them. It does not, however, supply a set of ''correct'' decisions for real-time or fault-tolerant systems. Unfortunately, such a set of decisions will only maintain its ''correctness'' when considered in the context of an application-specific environment. Instead, guidelines are provided on how to make the decisions for a more conventional, non-real-time application. Practitioners working in specialized domains may choose to make each design tradeoff differently than suggested in this paper, but the tradeoff analysis and discussion presented here will still be applicable.

2. A MODEL OF A REUSABLE SOFTWARE COMPONENT

It is important to begin any discussion of the topic of software reuse with a common understanding of the subject matter and terminology to prevent miscommunication. To that end, the following definition advocated by Will Tracz is proposed: "reusable software is software that was *designed* to be reused" [Tracz90b][emphasis added]. While this definition is somewhat circular, it does distinguish reusable software from "salvaged" software scavenged from existing non-reusable code, and from code "carried-over" from previous versions of a product [Tracz90b]. In addition, it highlights the fact that reuse is something that must be considered during *design*, not retroactively patched on to existing products as an afterthought.

Given this definition of reusable software, many new questions arise:

a. What does a reusable piece of software "look like"?

b. What are its characteristics?

c. How is it "tailored" for a specific application?

To answer these questions, and also to provide a common foundation for the discussions in the remainder of this paper, this section introduces a conceptual model of a reusable software component. A reusable component is more than "a piece of software"—it is an encapsulated abstraction, and as such it constitutes the basic building block for software reuse.

2.1 THE 3C MODEL

The "3C Model" of reusable software components was developed at the "Reuse in Practice" workshop, held from July 11–13, 1989 in Pittsburgh, Pennsylvania, by the Implementation Issues working group chaired by Will Tracz [Tracz90a]. It is based on Goguen's work with LIL[Goguen84a] and OBJ[Goguen83a], which has a sound mathematical foundation in category theory and many-sorted algebras. The name of the model comes from the three ideas upon which it is based:

a. The *concept*—what abstraction the component embodies.

b. The *content*—how that abstraction is implemented.

c. The *context*—the software environment necessary for the component to be meaningful.

To begin explaining these terms, consider a very simple and intuitive mapping of these ideas into Ada: the *concept* might become a generic package specification, each separate *content* might become a different package body for that specification, and the *context*ual decisions might be represented as the formal generic parameters of the package specification. Although this mapping is overly simplistic and hides many of the subtleties of the 3C model, it provides a more concrete image of the 3Cs in this initial discussion.

Figure 1 illustrates of how such a reusable component might be visualized. However, the three ''C'' terms are much more sophisticated than this mapping would suggest. Each will be discussed separately in turn. Note that throughout this paper, these terms will appear in italics to separate them from conventional usage of the corresponding English words.

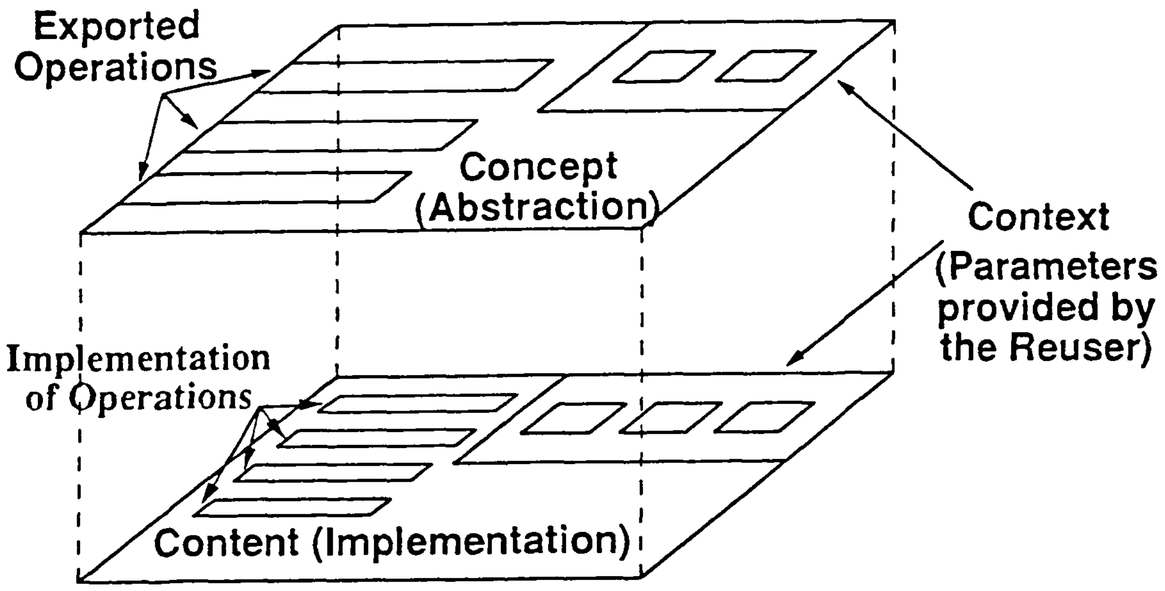

Exported Operations
Concept (Abstraction)
Context (Parameters provided by the Reuser)
Implementation of Operations
Content (Implementation)

Figure 1. The 3C View of a Reusable Component

The model begins with the *concept* that the component embodies. The term *concept* is used here to denote an abstract model of *what* the component does. This is the user's model of what can be done with the abstraction, rather than the component writer's model of how these capabilities are implemented.

A formal definition of the *concept* is very desirable, since it allows for tools that assure the user of a component that the component is being applied correctly. This definition may include the functional semantics of the *concept*, both for greater error checking and for verification purposes. An example of a reusable component's *concept* would be an Ada package specification augmented with Anna[1] to define its functional semantics [Tracz90a].

Note that although the *concept* encompasses the functional semantics of the component, the representation of the *concept* in a given language, such as Ada, is still useful even if it does not formally capture these semantics. The distinction between the *concept* and its *representation in a programming language* is vital—the *concept*, by definition, includes the component's functional semantics. A partial representation of the *concept* that does not formally capture semantics is still useful, however, because it does specify other aspects of the *concept*. The implementation of the component can still be automatically checked against those portions of the *concept* that are represented. The lack of functional semantics in such a representation will merely limit the amount of checking that can be done by machine to ensure that such a component matches the actual *concept*.

Next, the *content* of a component is the actual algorithm, the *how* which implements the *concept*. Of course, more than one algorithm may implement the *concept*, so more than one corresponding *content* is allowed. If the expression of the *content* allows (i.e., if the implementation language in which the *content* is defined allows), the *content* can be verified against the formal description of the *concept* to ensure it provides all "exported" capabilities.

An example of a group of implementations for a single *concept* in Ada is the collection of 26 stack packages found in Grady Booch's components [Booch87a,Tracz90a]. Such a "family" of implementations for a single concept, originally proposed in [Parnas76a], allows the user to select the implementation most suited to the task at hand.

Last, there is the *context* in which the *concept* and *content* are defined. The term *context* refers to those parts of the software environment external to the component that are relevant to the definition of the *concept* or *content*. This definition is a formalization of an intuitive idea of the "context" or environment in which a piece of code is defined and operates.

To gain a better intuitive understanding of *context*, consider a component author creating a new Ada package specification. This author might develop the specification "from first principles," using only the abstract machine provided by the language. No references to types, routines, or anything else not defined locally within the specification would be included. Such a package specification has no *context*—it does not use any external definitions. If, on the other hand, the author referred to any externally defined types, operations, packages, or objects, those external entities would be part of the *context* necessary for the definition of the specification. Similarly, the external definitions used in defining a component's implementation within an Ada package body are *context*. For a reusable component, all of these external definitions form the

1. Anna is an annotation language for Ada that, among other things, allows programmers to specify the semantics of Ada operations [Luckham85a].

context of a component, and they encompass *exactly what may change each time the component will be used in another application.*

By defining a clear boundary between what is in the *concept* of a component and what is in its *context*, the component writer can isolate change away from the core of the component. Then a potential reuser can configure the component for a given application simply by providing new values for the *context*. For example, in Ada the component writer could represent the *context* using generic parameters, while the *concept* would be represented by the remainder of the package specification.

The "parameters" that represent the *context* describe an abstract interface between the component and its environment. Ada generic parameters, for example, describe an interface to reuser-supplied types, objects, and operations, rather than an interface to a permanent fixture in the environment. And Just as a formal description of the functionality exported by a specification aids in error checking, a formal description of this *context*ual interface is also beneficial. Formal descriptions of the parameters to a component can provide a basis for determining if the parameter values supplied by a reuser match the requirements of the component.

In this paper, the term *conceptual context* will be used to refer to the *context* of a particular component's *concept* (for example, the formal parameters of an Ada generic package specification), while the term *implementation context* will be used to refer the the *context* of a particular implementation of a component's *concept* (for example, the withed units in an Ada generic package body). The categories of *context* can be further divided based on additional criteria, but this rough categorization is all that is necessary for this paper.

In order to get a more intuitive idea of what *context* really is, consider the following scenario. A component writer is constructing a reusable component embodying the abstraction of a "stack." In order for the *concept* (which, for example, may be represented as an Ada generic package specification) to be meaningful, the normal operations of **push** and **pop** must be defined to work on some data type. The definition of this data type is part of the *conceptual context*. The component writer may choose to make this data type a parameter of a generic package specification, leaving the choice of type up to the potential users. This approach of deferring decisions about a particular element of the *conceptual context* is shown in Figure 2.

Alternatively, the component writer may decide to hard-wire in a specific data type in his specification. "Binding" a particular element of the *conceptual context* when the component is written is illustrated in Figure 3.

Superficially, there is a similarity between *context* and parameterization. However, the discussion of Figures 2 and 3 indicates that some *context*ual decisions are made by the implementor, and are thus "fixed" choices from the point of view of a potential reuser. Other decisions are

```
-- to define this unit completely.
package stack_of_T is

        type stack is limited private;

        procedure push(the_item              : in     T;
                      on_the_stack          : in out stack);

        procedure pop(the_stack             : in out stack;
                     the_top_item          :    out T);

        . . . -- [remainder of stack operations omitted here]

end stack_of_T;
```

Figure 2. Simple "Stack" *Concept*

```
with Element_Types; -- This is still "context" because it is external
                    -- information which is necessary to define this unit
                    -- completely. In this case, however, the component
                    -- author has "hard-wired" the value of the context.
package hard_coded_stack is

        subtype T is Element_Types.user_defined_type_1;

        type stack is limited private;

        procedure push(the_item              : in     T;
                      on_the_stack          : in out stack);

        procedure pop(the_stack             : in out stack;
                     the_top_item          :    out T);

        . . . -- [remainder of stack operations omitted here]

end hard_coded_stack;
```

Figure 3. *Concept* **with Hard-Wired** *Context*

left "unbound" so the user may make them. This is in contrast with the traditional idea of "parameters," which are only "bound" by the end user.

Continuing the example scenario of creating a stack *concept*, as shown in Figure 4, imagine the component writer is now constructing an implementation for it. Note that the *context* of the implementation (environment of the package body) includes both the *concept* (the package specification) and the *conceptual context* (the generic formal parameters in the specification).

Now suppose the component writer chooses to implement the stack abstraction using a linked list for this particular *content*. The writer may employ yet another *concept*, embodying the abstraction of a linked list. The linked list *concept* would then be part of the *implementation context* for this implementation of the stack *concept*, a part which would be "fixed" from the point of

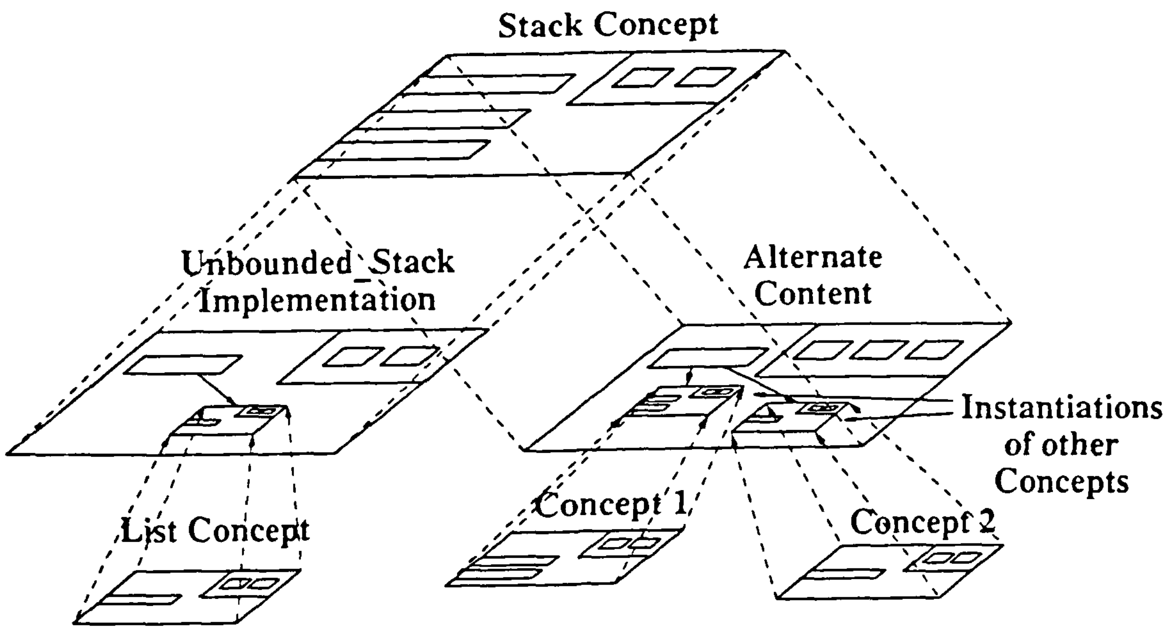
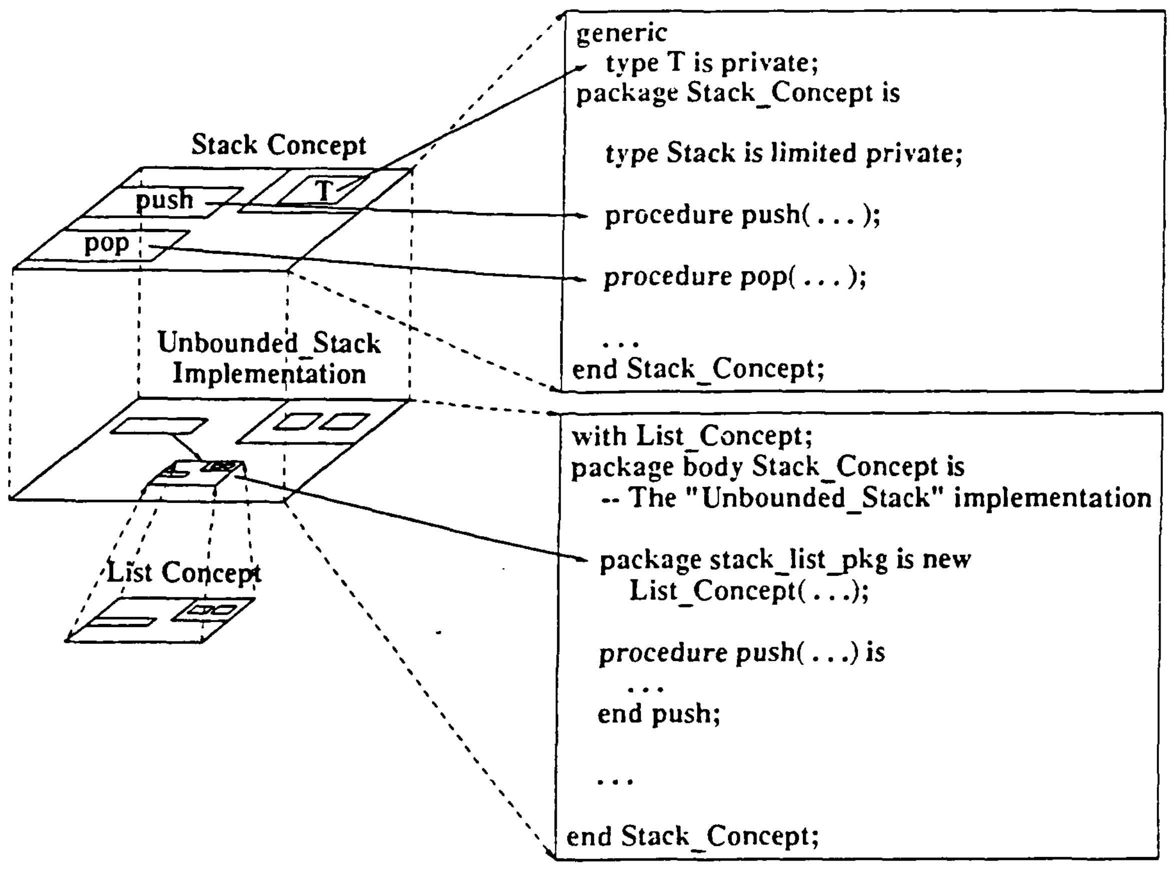

Stack Concept
Unbounded_Stack Implementation
Alternate Content
Instantiations of other Concepts
List Concept
Concept 1
Concept 2

Figure 4. A System of *Concepts* and Implementations

view of potential users. Suppose further that the linked list *concept* also has many possible implementations, varying in the way they manage dynamically allocated memory. The implementor may not want to restrict which implementation of the linked list *concept* is actually employed to represent stacks and may leave this decision, which is part of the *implementation context*, up to the user who would know more about what memory behavior is required.

Figure 5 shows skeleton Ada code for such a stack package and how this code maps into the structure pictured in Figure 4.

This example briefly illustrates the basic steps in creating a reusable software component. When creating such a component, it is important to first identify the abstraction which the component is to embody. This abstraction is formalized and becomes the *concept*. The algorithmic differences between particular implementations of this abstraction become separate implementations of *content*. The other differences (what type of data the abstraction deals with, numeric ranges, etc.) become *context*, which may either be bound by the implementor or deferred to the end user. In an Ada oriented environment, the *concept* might become a generic specification. Each separate *content* that the component writer chooses to implement might then become a different body for that specification. The *context*ual decisions deferred to the user might then become the formal generic parameters in the specification.



```
generic
   type T is private;
package Stack_Concept is

   type Stack is limited private;

   procedure push( . . . );

   procedure pop( . . . );

   . . .
end Stack_Concept;
```

```
with List_Concept;
package body Stack_Concept is
   -- The "Unbounded_Stack" implementation

   package stack_list_pkg is new
      List_Concept( . . .);

   procedure push( . . .) is
      . . .
   end push;

   . . .

end Stack_Concept;
```

Figure 5. Mapping the 3 "C"s into Ada

Defining reusable components in a programming language can thus be viewed as the task of separating *context* from *concept*, *concept* from *content*, and *content* from *context*. While this does not answer the question of how to design reusable components in general, it does provide a new perspective on the question of how to represent such components. Designing the component involves forming an abstraction and identifying the *concept* and the *context*, then separating them to achieve the best change control and reusability. Once this is done, one can concretely represent this abstraction in a given programming language.

2.2 THE PROCESS OF SOFTWARE REUSE AND ITS BENEFITS

With the model of reusable software components presented in Section 2.1 in mind, the question of where reuse provides savings can be considered. Intuitively, software reuse saves both money and time because it is presumably faster to "look up" a component in a library than to write it over from scratch. Charles Krueger, in describing how ineffective component retrieval systems can limit reuse, states this common thought as a simple requirement:

> To reuse a software artifact effectively, you have to be able to "find it" faster than you can "build it." [Krueger89a]

However, savings in the time to develop software is only a small part of the benefit which reuse can provide. It is well known that maintenance cost is the dominant term in the software life cycle equation [Parikh87a] [NBS84a] [Noel86a]. This fact implies that the cost associated with creating new code during development is much more than merely the time required to write it. New code also incurs maintenance costs. Further, the greatest potential for reuse benefits is in reducing maintenance costs.

To see how this potential may be realized, consider a well-established reusable component that has previously been used in many different applications. If this component is chosen for inclusion in a new system, it is likely to cost much less to test and debug because of its stability. In some sense, previous users of the component have contributed to the maintenance effort of the new system by debugging one of the components. Because this component is used in several delivered systems over time, it is possible for the total maintenance cost for that component to be amortized over all of the systems, including past, present, and future systems, that also employ this same component[2].

To illustrate how this cost sharing might happen, consider the best case scenario. In this scenario, all the systems using this reusable component obtain it from the same source. All of the maintenance teams in turn report all discovered defects in this component back to the source. In turn, the source updates the component (and also test cases, documentation, etc.), and notifies all of the registered users of the change. These users can incorporate the newer version of the component into their systems to eradicate the discovered defect.

Although this ideal scenario is far removed from current practice, examination of each step in the ideal scenario will show how maximal reuse leverage is achieved. Also, it must be pointed out that in current practice, this library would probably exist within the confines of a single project within a single company. Only one system would be involved, and the source of components would be a local module library. The component maintainers, component writers, and

2. The legal difficulties that may exist for cost amortization are beyond the scope of this paper.

component reusers would also be approximately the same group of programmers. The scenario presented here is phrased in slightly more grandiose terms to show how the central concept can be scaled to much larger reuse settings, but it is equally applicable to more common reuse experiences.

First, all of the systems in question must have the same component, and the easiest (but not only) way to achieve this is for the component to be supplied by a common source. The key is that all the systems must be using "the same component" in order for them to "share" the maintenance cost.

Note that when these costs are shared between systems, the sharing may not be equal. In particular, the first system to use a given reusable component may pay a much higher maintenance cost than a system reusing the same component after it had already been applied in many other software projects. These effects will depend on whether reusable components are certified or qualified some how before they are entered into a library. Additional techniques can also be used to try to equalize this burden if desired.

Second, the maintainers must report newly discovered defects back to the source. Without this feedback, defects are only removed from one system's local version of the component, and subsequent projects which obtain the component directly from the source will be maintaining code known to have defects. This will effectively prevent any significant maintenance leverage for software reuse.

The benefit of this centralized component source, or library, is clear—all future systems will have fewer defects to find. In addition, it is also possible for current reusers to benefit from fewer defects, if additional actions are taken. The "common source" can "feed forward" each new version of a component to the current reusers[3] of the previous version as defects are removed. This step is needed if current systems are to share maintenance costs with each other, although it is not necessary for realizing the primary benefit offered to future systems.

All of these steps in the ideal scenario also rely upon a strong version control system in the reuse library. It is vital that errors reported "from the field" be traced to the correct version of the component. Further, all future library clients should be given the latest, most error-free version of the component, even though older versions may still be maintained in the library because previous clients are still using them. It is also vital that each client of the reuse library use a strong configuration management system so that the added benefits of the "feed forward" approach can be obtained.

3. In this paper, the term *reuser* or the term *client* will be used to refer to a person who uses a software component, while the term *component writer* or *component author* will be used to denote a person who is constructing a software component for others to use.

In this scenario, only the defect correction role in maintenance has been discussed. Other maintenance tasks, such as fixing errors in the specification of the system, adding new functionality, or adapting the software to a different environment are also important. The modularity suggested by the 3C model addresses these maintenance tasks.

The 3C component model's emphasis on strict component boundaries and separation of concerns is aimed at promoting highly modularized components. A component's *concept* not only protects the reuser from implementation detail, it also protects the component developer from a reuser's expectations by completely defining the interface contract. This protection can be further increased by the automatic checking that can be done on *concepts* that contain semantic specifications.

This increased modularization helps to protect other components in a system from the local changes to a specific reusable component that result from other forms of maintenance. If a maintainer wants, this new version of the component can be submitted back to the original component's source as an "enhanced" version. The revised component may include additional functionality, an altered or improved specification, or the result of any other maintenance task. It can then be made available to other maintenance teams working on other systems using the original component in order to amortize the cost of other forms of maintenance, and also to the developers of new systems.

Thus, reuse of software components not only has the minor benefit of increasing productivity during development, but also has the major benefit of reducing maintenance. In addition, software reuse also leads to higher software quality; when some of a system's components have already been through the full life cycle on other projects, these components already have a long history of testing and usage that implies a much lower defect rate than that for new code. The bottom line is that the strongest reuse leverage comes from amortizing the maintenance costs for reusable units over all its "reusers," past, present, and future.

2.3 HOW COMPONENT TAILORING AFFECTS THE BENEFITS OF REUSE

With this leverage point in mind, it is important to examine how reusable components are constructed and tailored. Separation between *concept* and *content* follows the traditional lines of separating specification from implementation, which already exist in software engineering practices. This separation can often be achieved using the underlying separation mechanisms within the implementation language—for example, using Ada package specifications and bodies.

Separating out *context*, however, may not be directly supported by all implementation languages. When separating the *context* from either the *concept* or the *content*, it is important to know who will "bind," or provide values for, that *context*: the reuser or the component developer. Once the *context* has been identified, that portion of the *context* that will be provided by the reuser must be expressed in such a way that the reuser can actually control it.

When the reuser provides values for this *context* he is ''tailoring'' the component. There are a wide variety of mechanisms by which the user may tailor a component, each with its own associated method of describing the *context* that may be changed. A list of some of the n. ›st commonly used mechanisms would contain:

a. Source code modification

b. Simple text substitution

c. Simple preprocessing

d. Generic parameterization

e. Inheritance

   (1) Structural Inheritance—inheriting the structure of the component's *concept* (the interface)

   (2) Code Inheritance—inheriting the implementation within the component's *content* (the implementation code)

f. Application generators

This list of mechanisms, arranged roughly in order of increasing ''automated'' support, shows the extreme variability in possible approaches to tailoring a component. Although in principle any of these mechanisms can be used with any programming language, practical limitations may prevent this. Those mechanisms that provide the most automated support for tailoring are often costly to add to a language that does not already support them, and this cost must be traded off with the benefits of those tailoring mechanisms, which are already supported by the language.

Examining the costs and benefits of the available tailoring mechanisms can be very revealing. Tailoring affects both the component developer and the component reuser. For the developer, the costs of a mechanism are measured in terms of:

a. The difficulty of developing a tailorable component using that tailoring mechanism—how difficult is it to create a ''template'' that the reuser can configure?

b. The safety of the development process—how many ''template'' errors are automatically detected when the template is written?

For the reuser, costs are measured in terms of:

a. The difficulty of tailoring—how hard is it to provide the missing *context*, i.e., fill in the template?

b. The safety of the tailoring process—how many inconsistencies or errors are detected when the reuser performs the tailoring?

c. The difficulty of using the component once it has been tailored—does the structure of the template make it hard to use once it has been filled in?

The benefits of a mechanism are measured in terms of the generality that a developer can give to a component. It should be readily apparent that both these costs and benefits are highly dependent on how a given mechanism is supported by a given programming language.

When the maintenance leverage of reuse is considered, these costs and benefits can be seen in a new light. The benefits of a particular tailoring mechanism are not measured just in generality, but also in terms of how much it reduces the amount of required maintenance for a tailored component.

For example, source code modification might be considered a wonderful tailoring mechanism because using it will allow the ultimate in generality—all the reuser has to do is tailor a component ''enough'' and it will do anything. However, tailoring mechanisms that affect the maintenance requirements of the ''tailored'' component have drastically reduced benefits in terms of maintenance costs. The reuser may introduce quite a few errors into the component just by tailoring it. The maintenance team will then spend lots of time finding defects in the way the reuser tailored the component (rather than errors in the component itself). Notice that the cost for this maintenance cannot be amortized because the defects are specific to this particular tailoring. In addition, tailoring approaches like source code modification incur further costs in terms of the safety of component development and the difficulty of tailoring.

In fact, these observations lead to two interesting conclusions. First, if you cannot separate the defects in the tailoring, which result from errors in the *context* provided by the reuser, from the defects in the component, which result from errors by the component developer, you *cannot amortize maintenance cost for reusable components*. In addition, the key to increasing quality through reuse is to *prevent tailoring from introducing additional errors as much as is feasible*. Both of these conclusions imply that automated tailoring mechanisms with stringent error checking, both for the developer and for the reuser, are very important.

In addition to these conclusions, considering the maintenance leverage of software reuse reveals another important point introduced earlier. Both the costs and benefits of any specific tailoring mechanism are *highly language dependent*. The degree of automated support provided by the language can greatly affect these costs, in turn affecting the maintenance costs of tailored components.

2.4 POSSIBLE MISCONCEPTIONS ABOUT THE 3C MODEL

Although this introduction to the 3C model of reusable software is detailed enough for the technical discussions in this paper, in the interests of space and simplicity it does not cover the full depth of the model. As a result, it is possible that the reader may end up with some mistaken ideas about some facets of the model. While Ada examples are certainly appropriate for this paper's audience, seeing the model only through an Ada-oriented ''mapping'' may lead to misconceptions.

In order to preempt these misconceptions, this section lays out the most common misunderstandings that occur in learning the 3C model from an Ada perspective. Unfortunately, adequately explaining many of these misconceptions would require a full dissertation on the model. Instead, this section provides a brief explanation to forewarn the reader.

The first misconception is that a *concept* corresponds to an Ada generic package specification, and its *content* is expressed in the corresponding Ada package body. This mapping is simple and intuitive, but restricting. The *concept* is really an abstract model of what a component does, while an Ada package specification is only a description of the syntax used to invoke the services of a component.

Second, this simple Ada mapping often causes programmers to overlook the possibility of multiple implementations. A *concept* may have several implementations, all of which are selectable alternatives of the same component. Ada does not adequately support this idea, and the component writer must work around Ada's restrictions in order to supply multiple implementations. When these implementations are supplied as separate Ada packages, it is easy to consider them as separate components, as opposed to alternative implementations of the same component.

Third, using Ada might lead one to underestimate the necessity of the tailorable *context* for an implementation. Each implementation of a *concept* may need its own user-controlled parameters, which are independent of the *context* of its *concept*.

In the 3C model, the kinds of parameters used for a *concept* are the ones that are necessary to define the abstract functional model of the component. For a stack, for example, the data type that the stack holds is a necessary element of the *conceptual context*. For a given implementation of this stack *concept*, such as a dynamically sized array implementation, there may also be implementation *context* that is relevant only to this *content*. For example, if an array representing a stack grew and shrank by *increment* elements at a time, the size of the *increment* might be determined by the reuser when he tailors this component for his application. But this parameter is not part of the abstract functional model. Because Ada does not allow a package body to have its own set of generic parameters in addition to the parameters in the corresponding specification, programmers often ignore this aspect of the 3C model.

Fourth, it is very easy from the presentation in this paper to think of *context* primarily in terms of generic parameters, and as provided by the reuser. It is presented in those terms for simplicity, and because that is the best available mechanism for providing tailorable *context* in Ada. The idea of context is much broader, however.

In a more ideal language, a component writer may want to define one *concept*—one abstract model—in terms of another abstract model by difference. In that case, the other *inherited* abstract model would be part of the *context* for the new concept being constructed. In fact, it would be a part of the *conceptual context* that was bound by the component author, not tailored by the reuser. The term "inherited" is used here because inheritance is an ideal mechanism for defining by difference. This brief example shows that *context* is not only used by the reuser for tailoring, and that there are mechanisms other than generics that can be used to define the *context* and how it is bound.

In relation to the previous misconception, consider how a component writer might use implementation *context* to define a component's *content* by difference. One could use code inheritance mechanisms to define the implementation of some operations within that *content*. In fact, the implementation of these operations might be inherited from a completely different component than the one used in defining the *concept*. Inheritance in the *conceptual context* is used for defining the abstract model of *what* the component does, while a completely different inheritance hierarchy may be used in the implementation *context* to define *how* the abstract model is actually fulfilled.

This list of misconceptions is by no means exhaustive, but it does indicate the depths of the 3C model which are not presented here. Hopefully, this will help preempt "simplifying" misconceptions.

2.5 REUSE OF OTHER SOFTWARE ARTIFACTS

Now that many of the primary issues surrounding reusable code components have been brought out, the reuse of higher level software products such as designs, specifications, and requirements, will be briefly discussed. The current popularity of software reuse as a topic has led to several claims, some quite dramatic. One of the most seductive claims is that the "really big reuse payoffs" lie in reusing designs, or even higher level software abstractions, which are reinterpretable in different environments or applications:

> Design reuse is the only way we can come even close to an order of magnitude increase in productivity or quality. [Biggerstaff87a]

Such representations are inherently more general than code because they have less detail—fewer engineering decisions about them have been made that might rule out possible applications. Although this generality is obviously desirable, taking advantage of it can be tricky. Consider reusing a design in a different environment, for example reusing the design of a subsystem that was actually implemented in C under the Unix operating system and giving it an Ada implementation targeted for VMS.

In this case, the "really big payoff" is in terms of time to create the design, and *there is very little payoff in terms of reduced maintenance cost*. In fact, the only maintenance payoff will come from maintaining software developed from a "well tested" design, a benefit that is likely to be very small compared to the cost of maintaining all of the new code that will be written.

If, however, all of the software levels below the design were reused with the design (not possible in this example) maintenance payoffs could still be achieved. This would be analogous to reusing a software component that consisted of a very large subsystem. This might also be realized by embodying the design in an application generator, which would then generate source code corresponding to the subsystem. But these approaches may not be possible in all cases, and reuse at the design level alone certainly does give a payoff in terms of development time.

This does indicate, however, that reuse of higher level software representations is not the answer to all reuse problems, nor should code reuse be abandoned in favor of design reuse since it is the key to large payoffs during the maintenance phase of the life cycle. On the other hand, this discussion also indicates that reuse *should be considered at design time*, where opportunities for reusing the largest components are higher. If such large components can be reused as complete subsystems, there can be a very high payoff in both design and maintenance costs. This discussion is equally applicable to reuse of even higher representations, such as reapplying portions of specifications, and requirements.

2.6 REUSE OF MORE COMPLEX SUBSYSTEMS

In addition to the reuse of higher levels of software representation, the reuse of more complex software components, such as subsystems or systems, also raises issues. The most important issue is one raised during the Common Ada Missile Packages (CAMP) project [McNicholl86a], dealing with "coupling and cohesion."

One claim of structured software design that has an effect on reusable software is that modules or components should be loosely coupled and highly cohesive [Stevens79a]. While component designers may opt for newer alternatives to structured design, the interpretation of this claim within a framework of reusable software has clear benefits. Components are more reusable if they are independent of other components, and they are easier to reuse if they only encapsulate a single abstraction [Booch87a].

CAMP, one of the first industrial-strength projects faced with the problem of constructing reusable components, included the development of ''intermediate'' [Levy87a] components at the subsystem level, which were much more complex and domain-specific than the low level data abstractions found in typical component libraries. As a result of this effort, Sholom Cohen observed a curious characteristic of the CAMP intermediate components—they tended to be *highly coupled and loosely cohesive*[Cohen90a]. This observation opposes the intuitive idea that reusable components should be as independent as possible.

There are three potential causes for this problem, which the author has termed ''coupling inversion.'' First, the CAMP components were written for a real-time, embedded computer environment. The efficiency concerns present in such an environment may make certain tradeoffs between reusability and performance necessary.

Second, it is possible for a reusable subsystem-level component to be composed of tightly interdependent pieces. While these pieces may not be very reusable when considered separately, the subsystem they form may still be reusable. In other words, it is possible for a reusable ''whole'' to be composed of parts that are not reusable. Components of this nature, which may arise more often in the real-time arena, are discussed in Section 4.1.

Third, the CAMP program may have encountered the effects of the ''parameterization management'' problem. The CAMP components were written in Ada, using Ada's generic mechanism for tailoring. Naturally, as components grow to the subsystem level of complexity, there are more and more *context*ual decisions that the reuser can make. As the number of tailorable attributes grows, the costs associated with a particular tailoring mechanism, both in terms of how difficult it is for the author to set up the component and how difficult it is for the reuser to tailor it, becomes much more of a burden. At some point, it is possible that this cost may even outweigh (or be perceived to outweigh) the benefits of reusing the component. Specifically, it is possible that in the CAMP case, this point was reached in some subsystem-level components, and this contributed to the coupling inversion.

The parameterization management problem is an important concern. There is currently no model of the process for tailoring (or parameterizing), its purpose, what the available mechanisms are, or how the mechanisms are used. Instead, there is only a selection of mechanisms to use, without any real understanding of whether these mechanisms fill all the needs, or which ones are appropriate for what forms of parameterization. Section 5.3 will discuss this problem and current solutions more completely.

3. AN EXAMPLE OF A REUSABLE COMPONENT

In order to provide a common reusable component for discussion in the subsequent sections of this paper, this section will introduce an example written in Ada. Sections 5 and 6 will discuss different aspects of this component, adding additional detail to it as necessary. Thus, this section provides a high level overview of the chosen component rather than a complete discussion of all its characteristics.

Although the preceding sections do not require an in-depth Ada background, the remainder of the paper assumes the reader is already familiar with how the features of Ada are used to create software modules. In particular, the reader should be fluent in the use of all of the features discussed in Section 1, particularly Ada generics. This example does not show the elementary techniques of how one can "generalize" an application-specific code module or routine. Instead, it picks up where such elementary discussions leave off. The example is presented as a model of how such "generalized" components are written today so that the more advanced, Ada-specific issues which affect the component's reusability can be discussed. For the reader interested in the basics of using Ada's **generic** features to generalize software modules, [Tracz89a], [Mendal86a], and [Booch87a] offer excellent coverage.

The *concept* chosen to serve as an example is based on a "general purpose" data structure (GPD) developed for use at Ohio State University by the Reusable Software Research Group. This abstraction is a generalization of the idea of a "linked element" that can be used to create a wide variety of linked structures.

This abstraction was chosen as the paper's central example for several reasons. First, it is more sophisticated than the basic data structures most often chosen as exemplary reusable components. This is because *approaches that appear to work for simple data structures often do not scale effectively to the more complex structures used in larger software systems*. Second, the freshness of the example will hopefully make it more interesting for experienced readers. Third, the example is complex enough so that it can be used to discuss all of the points raised in this paper. On the other hand, it is still a single abstraction based on a data structure so that it can be presented and understood with minimal reference to other software modules. With these thoughts in mind, the remainder of this section presents an overview of the GPD *concept*.

An enhanced version of the GPD *concept* discussed in this section, supporting additional node categories and more complex functionality, is actually in use at the Institute for Defense

Analyses (IDA). While the component is a low level data abstraction, it is sophisticated enough to highlight many of the tradeoffs that must be made when designing reusable components. Sections 5 and 6 will discuss each reuse decision in detail, suggesting ways the specification presented in Figure 1 could be improved. Finally, Appendix C provides complete source code listings for the Ada package specifications of the GPD component, beginning with the "naive" version in Figure 1 followed by the progressively generalized versions which include the changes discussed in Sections 5 and 6.

3.1 DESCRIPTION OF THE GPD ABSTRACTION

Each element, or node, in a GPD structure has a "slot" that holds data of some user-defined type. In addition, GPD nodes can be classified by the way their outgoing links are structured.

Nodes that do not have any outgoing links are considered "leaf" nodes, while nodes which do have outgoing links are "non-leaves." Leaf nodes may contain an additional data slot in lieu of outgoing links, and are categorized based on this additional slot: *gpd_integer* nodes can hold an integer value in this slot, *gpd_boolean* nodes can hold a boolean value, and *gpd_empty* nodes do not have an additional slot.

Non-leaf nodes with outgoing links conceptually organized as an array are categorized as *gpd_parent*, and non-leaf nodes with outgoing links conceptually organized as a list are categorized as *gpd_sequence*. Notice that this is the *concept*ual model presented to the clients of the GPD component. The *content*, on the other hand, may use any appropriate method to implement this abstraction.

Some operations on GPD nodes are shared by all node categories, such as operations for accessing a node's commonly typed slot, or the deallocation operation. Each category also has a category-specific set of operations tailored toward manipulating its outgoing links.

Figure 6 provides an illustration of a very simple GPD structure. In it, Node 1 represents a *gpd_parent* node with an array of four outgoing links to Nodes 3, 4, 5, and 2, respectively. Similarly, Node 2 is a *gpd_sequence* node with a list of outgoing links to Nodes 3, 4, 5, and 1, respectively. Node 3 represents a *gpd_integer* node, Node 4 a *gpd_boolean* node, and Node 5 a *gpd_empty* node. This picture provides a more intuitive understanding of how the structures declared in an Ada package for this *concept* can be utilized.

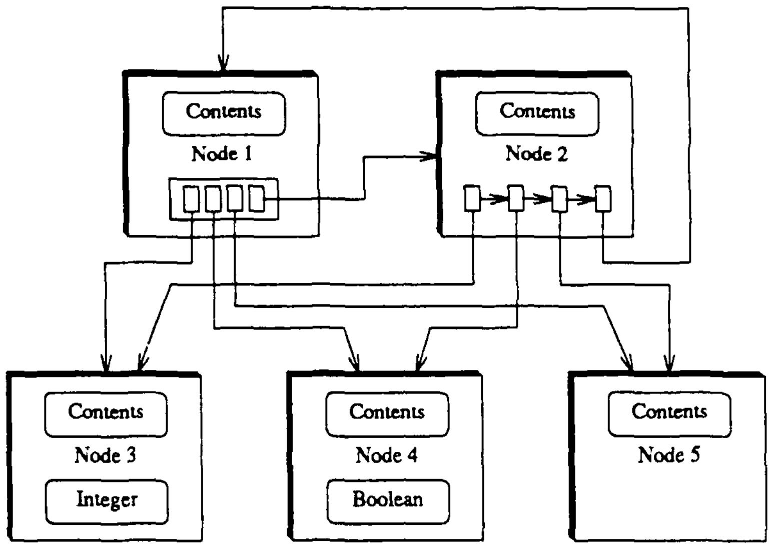


Figure 6. Graphical Depiction of a GPD Structure

3.2 THE ADA SPECIFICATION FOR THE GPD PACKAGE

This section discusses each piece of the Ada specification representing the *concept* of the GPD component. The full specification is provided in Appendix C for reference. This ''naive'' specification will serve as a starting point for the tradeoff analysis presented in Sections 5 and 6.

Throughout this paper, a consistent notation will be used when referring to the features of specific Ada code examples, or to specific Ada features. All Ada keywords will appear in bold face type. Similarly, all identifiers that appear in example code will appear in italics when they are discussed in the body of the paper. This will easily distinguish references to specific terms from the regular usage of English words that the terms may be named after. Note that in the comments within example code (which appear in italics), references to identifiers declared within the code will appear in all capitals. In addition, all Ada code that appears in this paper has been compiled and tested using a validated Ada compiler.

3.2.1 THE GENERIC PARAMETERS AND TYPE DEFINITION IN THE GPD *CONCEPT*

Figure 7 shows both the generic formal parameters of the GPD abstraction and the Ada type definition for *gpd_type*. The type *Common_Node_Contents* represents the type of information that is held in the primary data slot of every GPD node. As mentioned in Section 3.1, each *gpd_type* node also holds secondary information. The enumeration *node_class* defines the various flavors of GPD nodes, which differ according to the kind of information stored in this secondary

```
with text_io;
generic
        type Common_Node_Contents is private;
package GPD_pkg is

        type node_class is (gpd_empty,
                            gpd_integer,
                            gpd_boolean,
                            gpd_parent,
                            gpd_sequence);

        type gpd_type is private;
        null_gpd_node : constant gpd_type;

...
```

Figure 7. The Generic Parameters and Type Definition Used in the Specification for GPD *Concept*

location.

3.2.2 THE COMMON OPERATIONS SUPPORTED FOR ALL GPD NODES

Figure 8 shows the operations that the GPD concept provides for all GPD nodes, regardless of their *node_class*. These operations include those necessary to deallocate GPD nodes and reclaim the resources they are using, as well as routines to read and write the *Common_Node_Contents* slot of a given node.

3.2.3 AN EXAMPLE OF THE OPERATIONS DEFINED FOR LEAF NODES

Figure 9 shows the operations that are exported for a specific node class—*gpd_integer*. Notice that the category-specific operations for this node type are bundled into a subpackage within the main *GPD_pkg* specification. The operations for all other node classes are treated similarly. This divides the specification up into more manageable pieces to ease understanding. It also makes it possible for reusers to use the subpackage for the node classes they are concerned with, without extensively cluttering the name space within that Ada scope. Further, it allows users to use dotted notation to explicitly specify which versions of some overloaded functions are being used if they desire to increase readability.

All of the subpackages are similar in structure to the one pictured in Figure 9. The operations *new_node*, *get_data*, and *put_data* are all overloaded to increase usability of the package. Only one set of operations need be remembered in order to use any of the leaf nodes. The exceptions to this overloading approach are the *new_node* functions of the classes *gpd_empty*, *gpd_parent*, and *gpd_sequence*. Because the *new_node* functions of these three node classes are (or can be) called with no arguments, they are given distinct names so that there are not name collisions.

```
. . .
          -------------------------------------------------------------------
          -- The following 5 routines are common to all node classes.
          -- They include functions to determine the class of a node,
          -- deallocate a single node or a whole gpd structure, and
          -- read or write the COMMON_NODE_CONTENTS slot of any node.
          -- These 5 routines are followed by 5 subpackages, one for
          -- each node class. Each subpackage defines the node-class-specific
          -- functions for a give node-class. Note that some functions
          -- are overloaded (like NEW_NC DE, etc.) if the desired node-class
          -- can be determined from the arg ment profile, but ambiguous cases
          -- (like NEW_NODE for generating a gpd_sequence vs. a gpd_empty)
          -- are given distinct names so they do not have to be qualified
          -- with subpackage names.
          --
          function node_class_of(node : in gpd_type) return node_class;
          procedure free(node : in out gpd_type);
                  -- FREE is equivalent to recursively FREE'ing each child of
                  -- a parent/sequence, then using FREE_SINGLE_NODE. Nodes are
                  -- marked so that cycles in the GPD are handled correctly.
          procedure free_single_node(node : in out gpd_type);
                  -- This routine frees the space occupied by a single node.
          -------------------------------------------------------------------
          -- All gpd nodes contain an element of type COMMON_NODE_CONTENTS.
          -- These functions allow access to this component of every node:
          --
          function get_data(node : in gpd_type) return common_node_contents;
          procedure put_data(node     : in out gpd_type;
                             data     : in     common_node_contents);
. . .
```

Figure 8. The Common Operations Exported for GPD Nodes

```
          -------------------------------------------------------------------
          -- This subpackage defines the functions available for GPD nodes
          -- of class GPD_INTEGER. Each operation will ensure that its arg
          -- is of class GPD_INTEGER, raising GPD_ERROR if otherwise.
          --
          package integer_node_pkg is
                  function new_node(data : in integer) return gpd_type;
                  function get_data(node : in gpd_type) return integer;
                  procedure put_data(node    : in out gpd_type;
                                     data    : in     integer);
          end integer_node_pkg;
```

Figure 9. The Subpackage Defining the Operations for *GPD_Integer* Nodes

The subpackages for *gpd_empty* and *gpd_boolean* are not shown here because of their similarity to Figure 9. They are provided in Appendix C, however.

3.2.4 THE OPERATIONS SUPPORTED FOR *GPD_PARENT* NODES

```
----------------------------------------------------------------
-- This subpackage defines the functions available for GPD nodes
-- of class GPD_PARENT. Each operation will ensure that its arg
-- is of class GPD_PARENT, raising GPD_ERROR if otherwise.
--
-- A node of class GPD_PARENT has an ordered list of children.
-- From the user's point of view, this list is organized as an array. The
-- length of this list is determined by the parameter to
-- MAKE_EMPTY_PARENT_NODE when the node was first created, and
-- this size cannot be changed for that parent node. The
-- children (some of which may be NULL_GPD_NODEs, the constant
-- defined earlier in the package for use as a null value) may be accessed
-- in any order using their positions relative to the beginning of
-- the list (i.e., their array index). Indices run from 1 to
-- MAX_CHILDREN.
--
package parent_node_pkg is
    function make_empty_parent_node(
        max_children : in positive := 2) return gpd_type;
    function max_children(node : in gpd_type) return natural;
    procedure put_child(child_node     : in      gpd_type;
                        parent_node    : in out gpd_type;
                        position       : in      positive);
        -- This routine assigns the specified CHILD_NODE into
        -- the specified position of the PARENT_NODE's conceptual
        -- array of outgoing links. This overwrites any previous value
        -- there. Since objects of GPD_TYPE are represented as pointer
        -- values, this introduces structural sharing.
    function get_child(parent_node    : in gpd_type;
                       position       : in positive) return gpd_type;
end parent_node_pkg;
```

Figure 10. The Subpackage Defining the Operations for *GPD_Parent* Nodes

Figure 10 shows the subpackage that defines the available operations for *gpd_parent* nodes. A *gpd_parent* node is conceptually modeled as a structure that has two subparts: a slot for holding *Common_Node_Contents*, and a fixed-length array of slots for holding other gpd nodes. The size of this array is specified when *make_empty_parent_node* is called. New nodes are created with all entries in this array equal to *null_gpd_node*.

The function *get_child* can be used to read the value of any entry in this array. The procedure *put_child* can be used to set the value of any entry in this array. Currently, using the assignment operator on items of type *gpd_type* produces *aliases*, so that GPD structures can be shared between multiple references. This aspect of the package is discussed more thoroughly in Section 5.2.2. It is mentioned here so that the reader understands that using *put_child* inserts a reference to the *child_node* into the parent's conceptual array, potentially introducing sharing between different GPD structures.

3.2.5 THE OPERATIONS SUPPORTED FOR GPD_SEQUENCE NODES

Figure 11 shows the subpackage that defines the available operations for *gpd_sequence* nodes. Conceptually, this node class is very similar to the *gpd_parent* class, and all of the comments about the behavior of that class in the preceding section also apply here.

The primary difference in the *gpd_sequence* class is that instead of a conceptual, fixed-length array, this node contains a dynamically variable set of references to other GPD nodes that is conceptually arranged as a list. Upon creation, new *gpd_sequence* nodes have no entries in their list of children. New references can be added at either end of this list. Also, any entry in the list can be accessed by position, just as with the *gpd_parent_node*.

```
-----------------------------------------------------------------
-- This subpackage defines the functions available for GPD nodes
-- of class GPD_SEQUENCE. Each operation will ensure that its arg
-- is of class GPD_SEQUENCE, raising GPD_ERROR if otherwise.
--
-- A sequence node contains an arbitrarily long list of child
-- nodes, which may themselves be other sequences. These children
-- can be accessed, and the list of children modified, by the
-- subroutines in this package.
--
package sequence_node_pkg is
    subtype sequence_type is gpd_type;
        -- This subtype is just used for clarity in the
        -- declarations below to show where a node of class
        -- GPD_SEQUENCE is expected. If a node of a different
        -- class is used where this subtype appears, GPD_ERROR
        -- will be raised.
    function make_empty_sequence_node return gpd_type;
        -- Return a new GPD_SEQUENCE node with no outgoing links.
    procedure append(seq              : in out sequence_type;
                     new_element      : in     gpd_type);
    procedure remove_head(seq     : in out sequence_type;
                          head    :    out gpd_type);
    procedure prepend(seq              : in out sequence_type;
                      new_element      : in     gpd_type);
    procedure remove_tail(seq     : in out sequence_type;
                          tail    :    out gpd_type);
    procedure read_and_consume(seq          : in out sequence_type;
                               element      :    out gpd_type;
                               N            : in     positive := 1);
        -- Removes the Nth element of the list of outgoing links,
        -- placing its value in ELEMENT.
    procedure read_nth_element(seq          : in out sequence_type;
                               element      :    out gpd_type;
                               N            : in     positive := 1);
        -- places the value of the Nth element of the list of outgoing
        -- links in ELEMENT without altering the list.
    procedure consume(seq   : in out sequence_type;
                      N     : in     positive := 1);
        -- Removes the Nth element of the list, without calling
```

```
            -- FREE on the contents. The reference stored in that
            -- outgoing link is lost.
        procedure consume_n_elements(seq    : in out sequence_type;
                                     N      : in     positive);
            -- Removes the first N elements of the list, without calling
            -- FREE on any of the contents. The references stored in those
            -- outgoing links are lost.
        function length(seq : in sequence_type) return natural;
            -- returns the number of outgoing links
        procedure reverse_sequence(seq : in out sequence_type);
            -- reverses the order of the list of outgoing links
        function copy(sequence : in sequence_type) return sequence_type;
            -- produces a new node of class GPD_SEQUENCE with an
            -- identical list of outgoing links
        procedure concat(onto, from : in out sequence_type);
            -- remove all outgoing links from ONTO, concatenating them
            -- onto FROM's list of outgoing links. At completion,
            -- ONTO will have an empty list of links.
        function is_empty(seq : in sequence_type) return 'oolean;
            -- are there any outgoing links from SEQ?

    end sequence_node_pkg;
```

Figure 11. The Subpackage Defining the Operations for *GPD_Sequence* Nodes

3.2.6 THE REMAINDER OF THE GPD *CONCEPT*

```
  . . .

    -------------------------------------------------------------------
    -- Errors:
    -- This package only defines one exception, GPD_ERROR. This
    -- exception is raised whenever a node-class-specific function
    -- or procedure is called with an argument of the wrong class.
    -- The exception CONSTRAINT_ERROR is raised if NULL_GPD_NODE
    -- is passed into a routine.
    --
    gpd_error : exception;

private
    type gpd_block(class          : node_class      := gpd_empty;
                   top_size       : natural         := 0;
                   bottom_size    : natural         := 0);
    type gpd_type is access gpd_block;
    null_gpd_node : constant gpd_type := null;

end GPD_pkg;
```

Figure 12. The Private Part of the GPD Specification

Figure 12 shows the final portion of the GPD package. The **private** section provides the Ada compiler with information about the representation used for the type *gpd_type* for separate compilation. The *GPD_pkg* specification defines *gpd_type* as an access type to the incompletely declared type *gpd_block*. Thus, GPD nodes are physically represented as access variables, but the definition of the corresponding structure is deferred to the body of *GPD_pkg*.

Figure 12 also shows the single exception defined by the GPD abstraction. This exception is raised whenever operations that are only appropriate for nodes of a specific node class are accidentally invoked on a node of a different class. Also, the predefined exception CONSTRAINT_ERROR will be raised if any *get_data* or *put_data* operations are attempted on a GPD node that is equal to *null_gpd_node*.

4. BENEFITS OF ADA FOR REUSABLE SOFTWARE

The support Ada offers for "good software engineering practices" has already been discussed at great length in other publications [Watt87a] [Nielsen88a] [Booch86a]. Similar discussions of how Ada supports software reuse also abound [Mendal86a] [Gargaro87a] [Braun85a]. Rather than retread this ground, this paper will concentrate on how well the Ada language supports the model of reusable software components presented in Section 2. This model is the basis for discovering both the strengths and weaknesses discussed in this paper. Of course, as has been mentioned elsewhere, this model is still maturing. Knowledge of the benefits and limitations it points out will neverless be useful, even under alternative models.

4.1 SUPPORT FOR *CONCEPT*—PACKAGES AND LIMITED PRIVATE TYPES

The Ada language was engineered with the capabilities of abstraction, encapsulation, and information hiding in mind. The separation between specification and implementation in a software module, necessary to support these capabilities, can be enforced through the use of packages. Likewise, generics are an effective way of representing parameterized units. In fact, a generic package specification is a natural way to represent the *concept* of a reusable component in Ada.

Ada allows the construction of "opaque" type definitions, where the actual physical realization of the type is invisible to its users. Making a type exported by a package **private** restricts the operations available to clients of the package, preventing them from depending on a particular representation for that type. The only operations available to a client for such a **private** type are equality comparison, variable assignment, and the operations explicitly defined for that type by the component author[4].

Limited private types can be used to constrain "opaque" types even more. Exported **limited private** types do not even have comparison or assignment operators available—only the operations provided by the component author can be used on variables of this type[5].

Thus, Ada's primary means of creating abstract data types is through the declaration of programmer-defined data types, particularly **private** or **limited private** types. Each such abstraction may be encapsulated within a package, and this encapsulation is even required for **private**

4. Certain predefined attributes are also available for **private** types, as described in [DoD83a, Section 7.4.2].

5. Certain predefined attributes are still available for **limited private** types, as described in [DoD83a, Section 7.4.4].

and **limited private** types. The operations exported by this package then define the user's view of the abstract data type and its behavior, completing the abstraction. The use of **limited private** also allows all assumptions about how the abstraction is implemented to be completely contained within the corresponding package body.

Note that abstract data types defined in this manner can be used to represent either *passive* or *active* objects. Passive objects are like traditional data structures. Active objects, on the other hand, represent *independent agents* that execute concurrently and that can initiate actions on their own. Thus, active objects can be used to encapsulate separate threads of concurrent computation. Ada's tasking features are often used to represent such objects.

However, note that the *implementation* of an active object should not be visible through a component's *concept*. Whether each separate object of the abstract type is represented by a data structure, by a single task, by a group of intercommunicating tasks, by a combination of a passive data structure and a commonly shared server task, or by any other method is a detail best hidden in the *content*.

Together, the language facilities described above for creating abstractions form the basis for supporting the definition of component *concepts* in Ada.

However, just because the idea of a *concept* naturally maps into an Ada package specification, it is not necessary for *every* Ada package specification in a system to represent the *concept* of some component. This misconception may be suggested by the intuitive mapping between *concepts* and packages presented in Section 2.1, but notice that this mapping is the simplest of many alternatives.

In particular, a single reusable component may be constructed from an entire group of Ada packages. A single, high-level package can be used to consolidate the interface of this multi-package component so that the user only has one specification to explicitly deal with. This single package specification corresponds to the *concept* of the component, and its multi-package nature is a detail hidden within the component's implementation, or *content*.

This is an important distinction, since Ada's packaging features can be effectively used to address many problems other than simply describing *concepts*. It is certainly not the position of this paper that everything written as an Ada package *must* be a reusable component. In constructing real-time systems, for example, it is common for a collection of tightly interdependent packages or tasks to be constructed to perform some higher-level function. Because of their tight mutual coupling, it may not make sense to try to reapply these pieces independently of each other in another effort.

The real reusable component in such a system is the *collection* of tightly interdependent parts. If such a collection can easily be given a consolidated interface, it is a prime candidate for a reusable *concept*.

The result of this approach is the following two guidelines:

Guideline 1:

A *concept* should be represented as a single, generic package specification. All reusable components should be represented to the user in this way if possible. Even large subsystems should have a single point of visibility. Use subpackages within the abstraction to organize sets of related operations, if necessary, but maintain the "single top-level generic per component" mapping, even for components that are actually implemented using several packages. The user should be able to easily grasp the purpose/function of the abstraction, although it may take much more time to understand exactly how to fully utilize the supplied operations.

Guideline 2:

Each *concept* should provide one and only one abstraction—i.e., define a single object type. This will help to increase the understandability of the component, and also aid in separating pieces that may be independently reusable from one another.

Recall that these guidelines apply to reusable components, not to Ada packages in general. Section 5.1 will discuss some of the shortcomings of the language features presented in this section, and how to best use them for defining reusable components.

4.2 SUPPORT FOR *CONTENT*—PACKAGE BODIES AND MULTIPLE IMPLEMENTATIONS

As mentioned in the previous section, the *concept* of a component is naturally represented in Ada by a generic package specification, while the *content* maps into the corresponding package body. The separation enforced by Ada between a package specification and body supports the required separation between a component's abstract model and its realization in code.

In addition, by making this separation in Ada, there is a possibility of more than one "interchangeable" implementation of a given package specification, making a family of implementations feasible. It is possible for a package to have more than one body, with the programmer controlling which one is visible to the compiler at any given point[6]. In this way, the basic idea of multiple implementations for a single abstraction can be supported. This support has shortcomings, however. Section 5.2.1 will discuss these shortcomings, and how to overcome them when defining component implementations.

4.3 SUPPORT FOR *CONTEXT*—GENERICS AND WITH CLAUSES

Section 4.1 mentioned that Ada's **generic** features were appropriate for representing parameterized software modules. In fact, the generic formal parameters of a generic package are the most obvious choice for representing the *conceptual context* of a component. These parameters would then represent that portion of the *conceptual context* that the reuser could change in order to tailor a particular component to a given application.

In addition, those portions of the *context* that are bound by the component writer can be represented in Ada through **with** clauses. These clauses determine what library units are visible within a given package specification or body. Using **with** clauses, the component author can specify the fixed portions of both *conceptual context* (what an Ada package specification **withs**) and the implementation *context* (what a package body **withs**).

Together, these language features allow the component author to separate the *context* from both the *concept* and the *content*. Further, by choosing which feature is used to provide what *context*ual information, one can separate the invariant portions of the *context* from the portions that are designed to be bound by the reuser.

At this point, a definition of *coupling* and how it is affected by choices about *context* is appropriate. To explain coupling, as the term is used in this paper, consider two Ada packages, named *Pkg_A* and *Pkg_B*. Intuitively, one package is *coupled to* another package if it cannot be used independently of that package. In Ada, this situation can arise in two ways, as illustrated in Figures 13 and 14.

First, it is possible for the package specification of *Pkg_A* to **with** *Pkg_B*. This may be termed *horizontal* coupling, due to the illustration in Figure 13. This form of coupling indicates that the definition of *Pkg_A*'s interface requires information about *Pkg_B*.

Second, it is possible for the package body of *Pkg_A* to **with** *Pkg_B*. This may be termed *vertical* coupling. This form of coupling indicates that the *implementation* of *Pkg_A* uses *Pkg_B*.

Both forms of coupling have different advantages and disadvantages. Without any horizontal coupling, it would be difficult to define abstractions that had other abstractions as subparts, such as a stack of lists, for example. In addition, without vertical coupling, it would be impossible to construct larger packages that were implemented in terms of lower-level packages.

Unfortunately, horizontal coupling that is *fixed by the component writer and unalterable by the reuser* limits reuse. Rather than withing other abstractions that will be used in defining a

6. The mechanism that the programmer uses to control visibility of package bodies is not defined in the language, however. It is dependent on the tools available in the development environment.

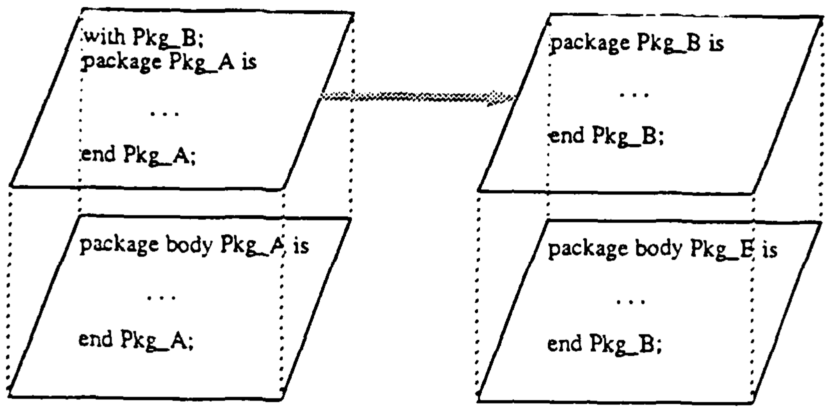


Figure 13. Horizontal Coupling

new *concept*, horizontal coupling should be routed through generic parameters if possible for greater flexibility. This leads to the following guideline:

Guideline 3:

> There should be *no* fixed, horizontal coupling between a *concept* and other *concepts*. In other words, Ada packages that represent reusable component *concepts* should not **with** other packages. Instead, all definitions required to describe the *concept* should be passed in through generic parameters.

Note that this guideline only applies to Ada packages that represent reusable *concepts*, not to all Ada packages. Inside a component that is implemented as a collection of Ada packages, those packages will certainly need to **with** other Ada packages to support the component writer's need for vertical coupling.

Unfortunately, practical limits may prevent strict adherence to this guideline. In particular, the efficiency cost associated with using generic parameters may be significant in some applications. Further, strict adherence to this guideline may lead directly to the parameterization management problem described in Section 5.3. If observation of this guideline is acceptable, however, it will lead to more flexible and reusable components.

Sections 5.1, 5.2, and 5.3 will all discuss the shortcomings of using generics to represent reusable components in this way. In addition, those sections will make recommendations on how to use the available language features to overcome some of these shortcomings.

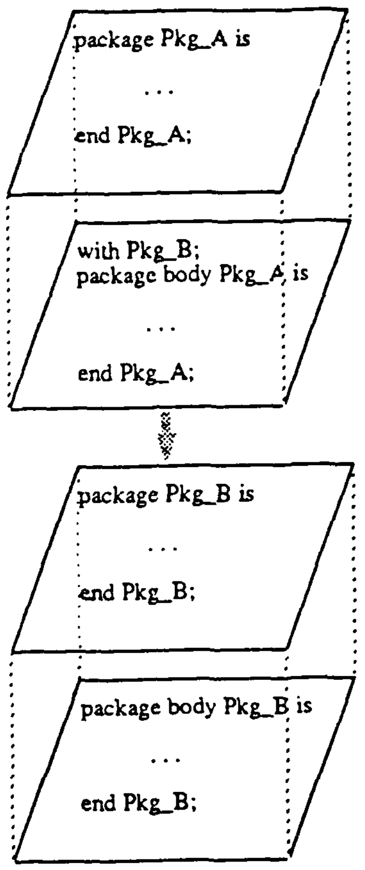


Figure 14. Vertical Coupling

5. LIMITATIONS OF ADA THAT RESTRICT REUSABILITY

This section describes several areas where the presence or absence of certain features in Ada impedes reuse. These features are interpreted in terms of the 3C model, and their ramifications on generality and usability are explored.

5.1 SUPPORT FOR *CONCEPT*—DATA ABSTRACTION AND ENCAPSULATION

To begin the discussion of abstraction and encapsulation, the following two definitions are offered:

a. *Abstraction* is the process of creating a higher level description that contains all the essential properties of some idea, but also suppresses all nonessential details. [Shaw81a]

b. *Encapsulation* is the process of collecting all of the necessary information about an idea or an abstraction in one location.

Both of these ideas are aimed at managing complexity and isolating change, and their benefits are widely known. In addition, they are the cornerstones of reusability, and of the 3C model. Suppressing unnecessary detail and placing it in a single location where it can be controlled is the key to managing change in a complex system. In this sense, unnecessary detail is not needed by the user of the abstraction, although it is relevant to the implementor. The goal is to create a single point of change for this detail in order to eliminate the "ripple effect" exposed changes can cause.

Ada supports both abstraction and encapsulation to a great degree, but does not require the use of either. This allows the component writer to make judgements about how much detail shows through an abstraction or how much encapsulation is used to localize changes. This section discusses the reuse ramifications of these tradeoffs in terms of the example presented in Section 3.

Ada's primary means of creating abstract data types is through the declaration of programmer-defined data types, particularly **private** or **limited private** types. Each such abstraction may be encapsulated within a package, and this encapsulation is even required for **private** and **limited private** types. The operations defined in this package thus define the user's view of the abstract data type.

For example, *GPD_pkg* encapsulates the GPD abstraction within a single package that is centered around the definition of the **private** type *gpd_type* and its associated operations. However, there are many places in *GPD_pkg* where there is insufficient abstraction (unnecessary detail, usually in the form of hidden assumptions, is given to the reuser) or insufficient encapsulation (the assumptions are not localized), which reduce the generality of the unit. Often, these faults are the result of poor design decisions, but a few are the result of failings in Ada. Each will be discussed in turn.

First, consider the *conceptual context* of the GPD component, which consists of the type *Common_Node_Contents*. This data type is provided by the reuser when tailoring (instantiating) the component. Figure 15 repeats the section of *GPD_pkg* where this generic formal parameter is declared. This code fragment represents the "interface" between the reusable component and the user-provided abstraction called *Common_Node_Contents*. While the form of generic parameter declaration presented in Figure 15 is in common use in the Ada programming community, it fails to provide a sufficiently abstract or sufficiently encapsulated description of the type *Common_Node_Contents*.

```
...
generic
      type Common_Node_Contents is private;
package GPD_pkg is
...
```

Figure 15. Declaration of *Common_Node_Contents*

While the declaration of *Common_Node_Contents* does hide much of the detail associated with the type, this detail is still a very significant part of the *context*ual interface. In other words, the detail is hidden, but not suppressed. The hidden details associated with *Common_Node_Contents* in this example include all of the "basic" operations defined for the type. Basic operations, sometimes referred to as primitive operations, are those operations that are implicitly defined in the Ada language for a given type. For this declaration of *Common_Node_Contents*, the basic operations include: assignment, comparison for equality, and an assortment of attributes[7]. In addition, this declaration carries an implicit assumption that the predefined versions of the basic operations—simple, structural assignment and comparison based on the physical representation of the type *Common_Node_Contents*—are appropriate for *all* instantiations of this component.

From the reuser's point of view, not only are these details hidden from view, they are also out of reach. There is no way for any of the basic operations on *Common_Node_Contents* to be redefined by the reuser, even if the assumptions implicit in the *context*ual interface will violate the

7. A complete list of the basic operations for any **private** type is given in section 7.4.2 of [DoD83a].

abstraction the reuser is trying to support.

Although one could argue that these details are all located in a single place (pictured in Figure 15), none of the details can be controlled from this location. In other words, this form of generic parameter declaration is also poorly encapsulated. Consider a component author who wants to change the interface to remove the assumptions of how the basic operations were implemented, for example. Not only will he have to rewrite Figure 15, the component author will also have to change the body of *GPD_pkg* so that it no longer depends on the basic operations and instead uses the new portions of the *Common_Node_Contents* interface.

More experienced Ada programmers will attribute these problems to an error in the design of the *Common_Node_Contents* of the component—this type should be **limited private**. In Ada, **limited private** types do not even have assignment or comparison implicitly declared as basic operations[8]. Thus, if assignment or comparison are needed on such a type, they must be explicitly declared as additional generic formal parameters that the reuser must supply as part of the tailoring process. Figure 16 illustrates how the *context*ual interface of *GPD_pkg* should be changed to convert *Common_Node_Contents* to a **limited private** type.

```
...
generic
    type Common_Node_Contents is limited private;
    with procedure assign(from : in      Common_Node_Contents;
                          into  : in out Common_Node_Contents);
    with function "="(left, right : in Common_Node_Contents) return boolean is <>;
package GPD_pkg is
...
```

Figure 16. *Common_Node_Contents* **as a Limited Private Type**

This alternative does provide a more abstract, encapsulated interface to the generic parameter *Common_Node_Contents* by explicitly capturing all of its basic operations in a user-controllable form. The problem is that Figure 16 captures all the basic operations that *Ada gives to the type*, not necessarily all of the primitive operations the user needs to support an arbitrary abstraction. In general, to model variables of an abstract data type, you must be able to create such variables that start off with valid values, move data into and out of such variables, and finally destroy the variables when you are done with them so resources can be reclaimed. While Figure 16 explicitly declares a basic operation for data movement (the *assign* operation), and the Ada language prevents the component writer from moving data into or out of a variable of type *Common_Node_Contents* without using this operation, there are no *initialization* or *finalization* operations defined in the interface. There is no way for the writer of *GPD_pkg* to ensure that any variables of type *Common_Node_Contents* are initialized correctly, or that such variables are finalized

8. A complete list of the basic operations provided for **limited private** types is given in section 7.4.4 of [DoD83a].

when he is through using them[9]. Moreover, the Ada language allows him to both create and destroy such variables even though these operations are available. The fact that Ada does not consider initialization and finalization to be primitive operations of a type is a failing that reduces its support for abstraction and encapsulation. Although component writers can define such operations in their interfaces, as demonstrated in Figure 17, the burden of enforcing their use within the body of the component is placed on the component writer, with no automatic means of error detection.

```
...
generic
      type Common_Node_Contents is limited private;
      with procedure initialize(data : out Common_Node_Contents);
      with procedure finalize(data : in out Common_Node_Contents);
      with procedure assign(from : in       Common_Node_Contents;
                            into  : in out Common_Node_Contents);
      with function "="(left, right : in Common_Node_Contents) return boolean is <>;
package GPD_pkg is
...
```

Figure 17. *Common_Node_Contents* **with Initialization and Finalization**

All of the points that have made about abstraction and encapsulation in the *conceptual context* of the GPD component are also applicable to its *concept*. This component is centered around the declaration of the abstract data type *gpd_type*, shown in Figure 18. Note that the basic operations provided for this type are implicit in the type declaration itself.

```
...
type gpd_type is private;
...
```

Figure 18. Declaration of *GPD_type*

As with *Common_Node_Contents*, this declaration hides essential detail from view while allowing that detail to affect every client of the GPD abstraction. In particular, Figure 18 implicitly states that the simple, structural versions of the assignment and comparison operators are to be used for variables of type *gpd_type*. Unfortunately, structural assignment on variables of this type produces an *alias* pointing to the original GPD structure rather than separate, independent copy of the GPD structure. Although the component author could provide his own semantically correct versions of these basic operations, he could not force the reuser to use them instead of the predefined versions if the type declaration stands as is. Once again, this leads to the use of a **limited private** type declaration so that the writer can enforce usage of the semantically correct operations.

9. Some *concepts* can be implemented safely without initialization or without finalization for some *contextual* parameter types. Such *concepts* either do not include the creation or do not include the destruction of values of such parameter types within their semantics. Thus, such *concepts* tend to export very simple functionality.

```
. . .
type gpd_type is limited private;
procedure assign(from : in      gpd_type;
                 into  : in out gpd_type);
function "="(left, right : in gpd_type) return boolean;
. . .
```

Figure 19. Limited Private Declaration of *GPD_type*

This modification leads to Figure 19. However, the type *gpd_type* still lacks basic operations for initialization and finalization. In other words, the component author has not provided a way for the reuser to create variables of the type *gpd_type* that start off with valid values. Once again, the component writer can supply these with the intent that they be used as basic operations. This configuration is illustrated in Figure 20. Unfortunately, there is no way the component writer can ensure that the reuser will actually *use* these basic operations. The writer trust the reuser to always initialize variables so that routines which operate on *gpd_type*s are not passed invalid values. Similarly, the reuser must be relied upon to reclaim resources via the finalize operation.

```
. . .
type gpd_type is limited private;
procedure initialize(data : out gpd_type);
procedure finalize(data : in out gpd_type);
procedure assign(from : in      gpd_type;
                 into  : in out gpd_type);
function "="(left, right : in gpd_type) return boolean;
. . .
```

Figure 20. *GPD_type* with Initialization and Finalization

Given that the above changes are made to the declarations of *Common_Node_Contents* and *gpd_type*, notice how significantly the remainder of the package must change[10]. Because all of the types are **limited private**, no assignment operator is available for either type. This rules out the elegant notation used in the original version of the package, presented in Section 1 of Appendix C. Because the type is **limited private**, the reuser does not have an assignment operator available for the type. Thus no functions returning the type can applied by the reuser.

This is a good example showing how increasing the generality of a component can decrease the ease of using it, even after it is tailored. A conscientious designer must carefully consider how the benefits in terms of increased generality trade off against the cost of actually reusing the component. Once again, the point at which the difficulty of use becomes high is language dependent.

Also note that because the *conceptual context* has grown in size and complexity, the component is more difficult to tailor. There are more parameters to provide, and more decisions to be

10. The complete text for this modified form of the package is presented in Section 2 of Appendix C.

made when a prospective reuser wishes to instantiate this *concept*. This is another tradeoff a designer must make, and is discussed more fully in Section 5.3.

The discussion in this section leads to the following guidelines:

Guideline 4:

Each abstraction should be *robust*, meaning that it should provide a *complete* set of basic operations. The client can only access instances of an abstract type using the operations exported by the component's *concept*. Therefore, the operations provided should be sufficient for the reuser to construct any complex manipulations that are needed from them.

Guideline 5:

For the abstract types defined in a component, use **limited private**.

Note that for some applications, and given the current maturity of Ada compiler technology, strict adherence to Guideline 5 may impose performance constraints that are unacceptable. In particular, some real-time applications may not be able to afford the extra cost associated with a procedure invocation for every basic operation.

If possible, such a situation should be addressed with judicious use of **pragma** *INLINE*[11]. Otherwise, the component designer has the option to use other type definitions. The primary disadvantage of ignoring Guideline 5 is that the abstraction is no longer strongly encapsulated, and changes can thus ripple through the code written by the reusers of such a component. Further, it limits the possibility of alternative implementations, although this is likely to be less of a problem in an such an application-specific problem area.

Guideline 6:

Always provide *initialize* and *finalize* operators for abstract types.

Again, designers concerned with efficiency may object to this guideline, particularly because *some* components can still be correctly implemented without explicitly exporting such operations. Designers concerned with the costs associated with these operations should consult [Harms89a]. Further, choosing to provide these guidelines on an "as needed" basis will make it more difficult for the user to reliably apply those operations to all types in a systematic manner.

[DoD83a, Section 6.3.2].

Guideline 7:

When writing a new component that uses other components, always faithfully apply the *initialize* and *finalize* operators. This guideline also applies to component reusers in general.

Note that some Ada 9X recommendations suggest that automatic support for this capability be added during the language revision process. This would alleviate the problems caused by relying on reusers to consistently apply Guideline 7.

Guideline 8:

All abstract types in the *context* (i.e., which are generic parameters in the package specification) should be **limited private**. Similarly, *initialize* and *finalize* operations for such a type should also be part of the generic formal parameter list. These operations should be consistently applied within the component's body.

Guideline 8 may also impose some performance penalties, given current Ada compiler maturity. The same objections mentioned above for Guideline 5 are often voiced for this guideline. Unfortunately, the cost of ignoring Guideline 8 is more significant—the resulting component will be less reusable. Certainly, in some applications it is desirable, or even mandatory, to trade off reusability for efficiency. However, the component designer should realize this trade off and make it consciously, rather than consistently ignoring Guideline 8 and thus making the tradeoff implicitly.

In addition to the above guidelines, the following suggestions are offered as a simple means of testing the generality of a component's interfaces. Certainly these suggestions are best suited to smaller components that export simple data types. However, they can be also be applied to larger abstractions.

Guideline 9:

As a test of the robustness of both the generic parameters and the exported operations of a component that defines an abstract data type, consider ''composing'' the component with itself. For example, you should be able to create a ''stack or stacks'' simply by taking the exported type and operations from one stack instantiation and using them to instantiate the same generic again. There is not a general requirement for this capability, but it is nevertheless a useful way of testing the robustness of both the generic parameters and the exported operations.

Guideline 10:

To promote this composability and a uniform view of abstract types, *all* types should match the following minimum profile:

```
type Item is limited private;
procedure Swap(left, right : in out Item);
procedure Initialize(i : in out Item);
procedure Finalize(i : in out Item);
procedure Copy(from     : intem;
               into     : in out Item);
function Is_Equal(left, right : in Item) return boolean;
```

Figure 21. Minimum Operations for Generic Formal Type Parameters

Note that the duplication operation is called *copy* instead of *assign* to highlight the fact that it may be costly, rather than the fact that it can be used to move data. The names actually given to these operations is of secondary importance, however. It is the functionality provided by these operations, as well as the number and placement of arguments, that is important for composability.

Although *copy* and *is_equal* are not primitive operations, they are included because in the cases where they are needed, the extra cost of constructing them from the primitives without access to the underlying representation is often prohibitive, as discussed in Section 6.3. Need for the *swap* operator will be discussed in Section 5.2.2.

5.2 SUPPORT FOR *CONTENT*

5.2.1 Multiple Implementations

Now that the idea of a reusable *concept* has been discussed, the possibility of a family of implementations for that *concept* can be presented. Initially though, one might ask what "multiple implementations" of a single *concept* are and why they are useful. Consider the GPD *concept* introduced in Section 3. It defines the *functional behavior* of a software module without specifying how the type *gpd_type* or any of its associated operations are actually realized in software. There are many possible Ada package bodies, *all with the same functional semantics*, which could be used to implement the GPD abstraction.

Although all the various implementations, or *contents*, for this concept are semantically equivalent, they still differ from one another in subtle ways. In particular, they differ in how they utilize resources: processing time, memory, and other application specific resources. The idea of "multiple implementations" gets it power from the fact that all these versions of the *content*, are "interchangeable" from the point of view of functionality. The programmer can choose the one that has the best resource utilization characteristics for the job at hand.

The idea of a family of implementations for the same *concept* differs from the idea of a family of related *concepts*. In a family of closely related *concepts*, each variant is an extension of some core *concept*. Each variant represents a different combination of extended capabilities. For example, the following GPD abstractions are a family of *concepts*:

a. The GPD facility shown in Section 1 of Appendix C.

b. The same facility, supporting operations to save and restore *gpd_type* variables to and from a file.

c. The initial GPD facility extended to support persistent GPD structures that continue to exist from one program invocation to another without requiring the user to save them to a file or restore them.

These three abstractions are all based on the basic idea of the GPD structure. However, each one is a different *concept*—they each provide a slightly different set of operations, or slightly different behavior, so they are *not* functionally equivalent. In other words, the three alternatives presented above each provide a different abstract model to the user.

Given this definition of multiple implementations, consider how such modules might be implemented in Ada. The simplest way is just to provide several versions of the package body *GPD_pkg*. Because Ada enforces the separation between package specifications and bodies, this can easily be done. There are two significant problems with this approach, however. Both of them are limitations of the Ada language.

First, Ada requires the representation for all externally visible types to be declared in the package specification. For **private** and **limited private** types, such as *gpd_type*, this means the physical representation of the type must be declared in the **private** section of the specification[12]. This private part limits all package bodies that correspond to that specification. It is a common trick when declaring a **private** or **limited private** type to specify the type as an access type within the **private** part. This allows the actual representation to be deferred to the package body using an incomplete type specification [Muralidharan89a] [DoD83a, Section 3.8.1]. In Section 1 of Appendix C for example, *gpd_type* is declared as an access type to *gpd_block*, which is given an incomplete type specification. The physical representation of the type *gpd_block* is deferred to the body of *GPD_pkg*, and if there are multiple bodies it may be different in each one.

12. Some consider the private part to contain implementation detail that hinders reuse [Muralidharan89a].

Second, at link time Ada only allows one body to exist for each package specification. Even if multiple implementations are provided, only one can be linked into an executable. Thus all clients of a given package within a single Ada program must use the same implementation of it.

All of these points apply equally well to generic packages in Ada, although generics make the lack of a solution more debilitating. Even if there are multiple implementations for a given generic package, a single implementation must be chosen *for all instantiations in a single program*.

In addition, generics raise other problems with multiple implementations. For most practical purposes, recompiling the body of a generic Ada package requires the same amount of recompilation as if the corresponding specification were also recompiled. Virtually all current Ada compilers implement generics so that all instantiations of a generic must be recompiled when the body of the generic changes. This has important implications if switching package bodies will be used as the method of selecting among alternate implementations.

With these points in mind, only one general-purpose solution is evident: to achieve multiple implementations in Ada, create a separate package specification/body pair for each implementation, giving each pair a different package name. Make sure all the specifications look the same. Then changing implementations can be accomplished simply by altering the package name in the **with** clause where the component is brought into scope. Although this idea may offend some purists, it is the most pragmatic approach to the problem to date.

When implementing this solution, version control of a large number of "parallel" packages can be a problem. For proper version control of the multiple implementations, one may want to use a preprocessor, generating all the package specifications from the same source. This approach might also be used with the package bodies to ensure that common code segments come from a single source. Alternatively, the common code can be separated out into generics which are used across several implementations of the same concept. This approach, both for code sharing across multiple implementations and sharing across families of *concepts*, is nicely demonstrated in [Musser89a].

In the future, these problems may be solved in several ways. Proposed Ada 9X modifications suggest adding support for multiple implementations within the same executable to the language. Alternatively, a module interconnection language for Ada, such as LILEANNA [Tracz90c], that offers these capabilities may be used.

Also, note that Ada does not support the separation between *conceptual context* and *implementation context* described in Section 2.1. By allowing the package bodies of generic packages to have their own generic parameters that are independent of those on the specification, this support could be added. This would be very useful for directly representing *implementation context*, allowing more of the the abstract model of the component to be represented directly in the language.

Because of this limitation, those portions of the *implementation context* that the component writer wishes to place under the reuser's control should be included as generic formal parameters of the corresponding specification. Since each implementation will have it's own specification, this will not interfere with the parameters that are shared by all versions of the component.

The following guidelines summarize the approach to providing multiple implementations for the same reusable *concept*, as presented in this section:

Guideline 11:

Each implementation of a *concept* should exist as a separate Ada generic package. However, all the package specifications for these implementations should be identical except for the package name. Also, these specifications may have additional generic parameters added that represent parameters to the corresponding implementation. These *implementation context* parameters, of course, are not necessarily uniform across all of the implementations. Thus, the Ada specifications may also differ in this respect.

Guideline 12:

The Ada package specifications for multiple implementations of a single concept should come from a common source, for example, using a preprocessor. The Ada package bodies for multiple implementations should share common code. Use lower-level generics (see [Musser89a] for an example), or a preprocessor so that common code comes from a single source.

5.2.2 Data Movement

> Although we use limited private types for all our monolithic components, the generic formal type denoting the item of a structure is typically private, not limited private. Why is this the case? By requiring a match with a private formal type, we are asserting that assignment of objects of the type is predefined. *If we required an explicit copy, the computational expense for using a structural component would be high* [emphasis added]. [Booch87a, page 581]

This quote illustrates one of the biggest objections programmers raise to the techniques suggested in Section 5.1—they are "too inefficient." This concern is particularly acute in the real-time community. This section discusses the root of this problem as well as how it can be overcome.

As Booch has pointed out, the inefficiency is a direct result of the "copy" operation, which operates on the generic formal type. Intuitively, it seems that such an operation would only cause a performance penalty *when it was used*. Unfortunately, in Ada there is an underlying assumption that *assignment* (with "copy" semantics) is the primary method by which data is moved from place to place. In order to get a value from variable A to variable B you have to use some form of variable assignment, which copies the value from one location to the other. But how does this impact reuse?

A "copy" operation involves more work than is initially apparent. Consider "copying," or duplicating, an object present at location A into another location called B. This involves three steps: deleting any object that might be present at B, creating a new object at B to hold the same information as A, and, finally, duplicating the information at A in the newly formed object at B. If the object type under consideration takes N memory units to store, then each of the three operations generally requires $O(N)$ units of time. For "trivial" objects, like integers, the first two steps do not involve any action at all and the final step is very brief because N is so small. For an encapsulated data object, such as a stack or a GPD structure, however, all three steps are not only expensive but also required.

Because of this, it is apparent that the cost of true "copy" semantics, i.e., duplicating the state of an object, is ordinarily linear in that object's size. Further, consider what happens if a generic reusable component has such an object type as a parameter and requires a copy operation for it. *Nothing can be said about the performance of that generic independently of its instantiation parameters*. In fact, if the user is considering using this component with nontrivial data structures, the performance penalty of true "copy" semantics may even be too high for this component to be reused. For this reason, the assumption that "copy" is the primary data movement operation is another of Ada's failings, although it follows in the footsteps of most traditional languages.

As an example, consider a stack generic, such as that introduced in Section 2.1, with a generic formal parameter specification as suggested in Figure 17. If a user wanted to create a "stack of GPD structures" using this generic, he would have to provide a copy operation for GPDs. The cost of copying a GPD each time he pushed it onto the stack might be prohibitive, however. In fact, this design might make the stack component all but useless for nontrivial situations!

There are several alternatives that overcome this efficiency problem. As one possible solution, the "copy" operation can be optimized so that it is more "efficient." Alternatively, the ''copy'' operation can be replaced with a ''sharing'' operation so that the same object can be referenced in many places rather than duplicated. Finally, it is possible in many cases to *eliminate* the copying altogether.

5.2.2.1 Optimizing Copy Operations

First, consider the prospect of optimizing the ''copy'' operation. The use of the word optimization here denotes a ''safe,'' semantics preserving transformation. The most effective copy optimization, known as ''copy-on-write,'' is often found in virtual memory systems [Tevanian87a]. The premise behind this approach is to postpone the actual duplication of the object as long as possible. This technique is often called lazy evaluation.

If the object at location *A* were being copied to location *B* using lazy evaluation, initially, the object would not be duplicated at all. Instead, a ''virtual'' copy of the object would be placed at *B*. This virtual object would point to the actual object located at *A*. This single object would be *shared* at both locations, although there would appear to be two distinct objects. No duplication would be required until one of the objects was changed by some form of *write* operation. At the last possible moment, immediately before either *A* or *B* were changed, the virtual object at *B* would be replaced with a duplicate of the object at *A*.

As evident from this example, lazy evaluation only increases efficiency if, at least some times, the duplication never has to be performed. Fortunately, because copying is most often used to move data values from one variable location to another, there is a significant potential for efficiency gains through this optimization. Unfortunately, however, lazy evaluation of copy operations is rarely implemented. There are two key problems with ''lazy copying'' that make it much less desirable to component designers.

First, if the copy operation for GPD structures were implemented using lazy evaluation, the change would be in the *GPD_pkg* body, not in the stack component. Lazy copying is an optimization of the copy operation itself, not of how the stack component uses the copy operation. Although installing this optimization in the GPD component will make ''stacks of GPD structures'' much more efficient, ''stacks of binary trees'' will still have the original efficiency problems. In fact, most *concepts* would have to be implemented using lazy copying in order to completely alleviate this problem.

Second, implementing lazy copying is a relatively complex task compared to the more conventional implementations of ''copy.'' It may add unjustified difficulty to otherwise straightforward *concepts*. While a low level generic that implements lazy copying can be constructed, many programmers would object to the added layers of procedure calling required. For all of the above reasons, lazy evaluation of copying is often ignored or specifically omitted. Of course, this need not be the case in actuality, but it is one reason often raised to excuse the absence of such an implementation.

5.2.2.2 Using Structural Sharing Instead of Copying

Now, consider the second alternative—structural sharing instead of copying. By adding an extra level of indirection through an access type, the "copy" operation can duplicate pointers rather than copying the object itself. This is a well known technique which programmers working in many languages use every day to reduce the overhead of assignment.

The disadvantage with this technique is that it results in "aliasing" problems that can lead to error-prone code which is very difficult to debug. These problems result from the fact that copying a reference is not the same as duplicating thc referenced object. Unlike lazy copying, the duplication *never* occurs. The same object is now pointed two from two different locations, leading to the idea of an object "alias." A *write* operation from either location affects the value seen from *all* locations. Because they are so difficult to find, these aliasing problems are often named as the largest class of software defects.

The biggest difference between lazy copying and structural sharing is that while lazy copying hides the details of when to share versus when to duplicate within the "copy" operation—structural sharing instead places this responsibility on the user of a component. It is up to the user to manage all the aliasing problems by deciding when to "share" and when to "duplicate." Usually, it is also the user's responsibility to remember whether the assignment statement for a given type actually duplicates data or introduces structural sharing.

Because the copy and aliasing operations are semantically different, components written *expecting* a copy operation, such as the stack component, *may have different semantics* if they are given an aliasing operation. In particular, operations such as *copy*, *clear*, and *is_equal* on the type *stack* may all have their semantics altered by the addition of a level of indirection and the use of an aliasing operator instead of a copy operator.

However, this is the approach that Booch and other authors have suggested to alleviate the efficiency problem. Unfortunately, components that place the responsibility of managing aliasing behavior on the user *are poorly abstracted and encapsulated*. The details of how operations are implemented to gain efficiency should be completely hidden within the *content* of a component, not placed on the user's shoulders.

This does not mean that structural sharing inherently violates encapsulaiion or abstraction, only that such sharing should be completely controlled within a component. As Booch notes, sharing is an important part of many useful abstractions. Relatively simple reference counting schemes are sufficient to hide this detail completely within a component[13]. Section 2 in

13. Such schemes do require the use of *finalization* operators, as suggested in Section 5.1.

Appendix C shows a version of the *GPD_pkg* specification for such a package. Note that structural sharing is still part of the GPD abstraction, although the sharing is only introduced through operations exported by the package, not by a built-in assignment operator. This allows the module to completely control all ramifications of the sharing, preserving the semantics of sensitive operations such as *clear*, *copy*, *is_equal*, and *finalize*.

5.2.2.3 Eliminating Copy Operations With Swapping

Now consider the final prospect of removing the "copy" operations completely. Are there other ways of moving data than copying? Douglas Harms and Bruce Weide suggest that "swapping" may be a more effective data movement operator in [Harms89b]. When two objects stored at locations *A* and *B* are "swapped," the object stored at *B* is not destroyed, no new object is created, and no duplication is necessary.

Where it is possible to use swapping instead of copying, swapping is a very big win in terms of efficiency. While the exchange of the two objects is apparently an *O(N)* operation just as copying is, it is much easier to optimize than "copy." By adding an extra level of access types as Booch suggests in the quote above, one can swap references to objects instead of swapping the objects themselves. Unlike copying, however, *swapping references is semantically equivalent to swapping the objects pointed to by the references*.

Also, note that the efficiency of swapping references to an object does not depend on the size of that object. Instead, it depends on the size of the reference. Assuming that all references are (approximately) the same size, *all* swap operations can thus be implemented in *O(1)* time!

Unfortunately, swapping cannot replace copying under all circumstances. The two operations are not "semantically isomorphic"; in other words, they are not equivalent in terms of what they can be used to do. Copying can be used to "create" entirely new objects that are duplicates of existing objects, while swapping can only be used to "move around" objects that currently exist. This means that there are times when swapping cannot be used instead of copying.

Fortunately, in most cases where copy operations are currently used, the actual *duplication* of objects is not needed—only object movement is required. This is why lazy copying works so well as an optimization; the duplication can be avoided most of the time because it is not actually needed. For straightforward components like stack abstractions, the most common operations are putting data into the structure, or retrieving data from the structure. Conventionally, copy operations are used for both of these tasks, when a swap would be just as effective. "Inserting" or "removing" a GPD structure from a stack makes much more sense than "duplicating" it into the structure and then "duplicating" it back out again.

Because swapping cannot fully replace copying, copying will still be required in some places. By choosing "swap" as the primary method of data movement within components, however, the clients of a component have to *explicitly* choose to use operations with copy semantics, rather than be faced with the performance penalties of copying in every operation.

Use of "swap" instead of "copy" as the primary data movement operator in this way addresses both the efficiency problem of explicit copying and the unnecessary complexity problems that structural sharing introduces. Although Ada doesn't provide "swap" as a primitive operation, component writers can export it for clients. Most importantly, it allows the construction of strongly encapsulated abstractions which are efficient *and* maintain their integrity rather than placing burdens on their clients. Figure 22 shows how the primary operations for a type exported by such a component might be declared.

```
...
type gpd_type is limited private;
procedure initialize(data : out gpd_type);
procedure finalize(data : in out gpd_type);
procedure swap(left, right : in out gpd_type);
procedure assign(from : in      gpd_type;
                 into  : in out gpd_type);
function "="(left, right : in gpd_type) return boolean;
...
```

Figure 22. ***GPD_type*** **declared with** ***Swap*** **in Addition to** ***Assign***

Note that changing to swapping as the primary basis for data movement affects many of the other operations exported by a unit. To examine this effect, consider the different types of operations available on a data type. Booch offers this taxonomy of operations [Booch87a, page 20]:

a. **Constructor**—An operation that alters the state of an object.

b. **Selector**—An operation that evaluates the current object state.

c. **Iterator**—An operation that permits all parts of an object to be visited.

Both constructors and selectors are implemented using a "copy" operator in a conventional language. When applying swap-oriented programming techniques, however, the most common form of operation is more aptly termed an *accessor*. An accessor allows the client to "swap" some portion of the object's state with the contents of some variable in the client's routine. By using the accessor to examine portions of the state and then to replace those portions back into the object, the client has the functionality of a selector. Also, the client could modify his local variable before swapping it back into the object, or use a completely different source for the object state, to achieve the functionality of a constructor. All of this can be done without resorting to copying, resulting in reusable components which are efficient regardless of how they are tailored (instantiated).

Section 3 in Appendix C contains a version of the GPD component rewritten to take swapping into account. The major differences are in the following operations:

a. *access_contents* (previously, *get_data* and *put_data*).

b. *integer_node_pkg.access_data* (previously, *get_data* and *put_data*).

c. *boolean_node_pkg.access_data* (previously, *get_data* and *put_data*).

d. *parent_node_pkg.access_child* (previously, *get_child* and *put_child*).

Finally, there is the question of whether a "copy" operation for an abstract type should be provided at all. Initially, it appears that one is necessary, since swapping cannot be used to build a "copy" operator. Surprisingly, a "copy" operation on an abstract data type is often *not* required. If accessors for all of the sub-parts of an abstract object's state are provided, and copy operations are available for all of these subparts, a "copy" for the composite object can be constructed. More importantly, such a "copy" operation built from the available primitive operations is often as efficient as a "built-in" copy operator—within a constant factor for the overhead of procedure calls.

In [Weide86a], the following distinction is made for such operations:

> Some operations will be essential (*primary* operations) and others merely convenient (*secondary* operations), in the sense that the latter could be implemented using the former. Usually, you should not include a secondary operation in a basic facility unless you anticipate that its implementation in terms of other operations will suffer more than a constant factor performance degradation from a realization that has access to the representations of variables.

While this recommendation is desirable, in practice it is often important to include the secondary operations *copy* and *is_equal* for an abstract data type. The reasoning behind this exception is given in detail in Section 6.3.

5.2.2.4 Guidelines on Data Movement

As a result of this discussion on data movement, the following guidelines were developed:

Guideline 13:

> Aliasing behavior (structural sharing) *is the responsibility of the abstraction, not the user!* "Structural sharing" semantics are often useful, but *all* basic operations must maintain the same semantics (in particular, *finalize*). Users should not be able to create aliases in an uncontrolled way (say, through use of the built in assignment operation). Instead, they may only call operations in the abstraction, which will then create aliases on their behalf (i.e., the package/abstraction must *always* maintain control over structural sharing).

This guideline is somewhat controversial, particularly among programmers concerned with efficiency. Many of the objections raised to Guidelines 5 and 8 are also raised here. However, another significant objection applies to Guideline 13.

Adhering to Guideline 13 may require a much more complicated implementation. This is of particular concern for component writers who are trying to write new components as part of a current development effort. The extra cost involved in following this guideline may not be justifiable within the context of that project's needs.

However, components that not only support but encourage explicit, uncontrolled aliasing by the reuser are *more expensive to reuse*. Such components encourage, or even require, the reuser to use them in ways that are error-prone, as mentioned in Section 5.2.2.2. Rather than place the burden of debugging such behavior on every reuser, Guideline 13 suggests that this behavior be captured and controlled within the component. Although the component may be more expensive to create and debug, the goal is to eliminate these bugs *in one place*. Then, all of the future reusers can benefit from it rather than duplicating the effort.

There are certainly cases where it is more pragmatic to ignore Guideline 13. However, the component designer should understand the reasoning behind this recommendation before he chooses to do so.

Guideline 14:

> Do not use the built in assignment operator as the basic data movement operator. Do not replace it with a *copy* operation. Instead, use a *swap* operation.

As indicated in Section 5.2.2.3, consistently using the assignment operator for moving values from one location to another *is not scalable to large abstract data types*. In particular, many of the efficiency concerns expressed about reusable components are exacerbated by this approach. Guideline 14 is aimed at directly addressing this problem by providing an alternative option that *can* be scaled to large structures without efficiency penalties. Although this is a nontraditional approach, designers who choose to bypass this guideline should understand how reuse of their components will be affected by efficiency concerns.

Concern over producing components that do not suffer efficiency penalties for dealing with large abstract data types is also reflected in the following two guidelines that support the concepts developed in Section 5.2.2.3:

Guideline 15:

Every component should define a *swap* operation on its abstraction.

Guideline 16:

All operations Booch would classify as "constructors" or "selectors" should be designed using "swap" semantics, not "copy" semantics. The one name/one object paradigm Booch uses is the correct approach, although assignment/copy is the *wrong* underlying data movement primitive.

The final guideline suggested as a result of Section 5.2.2 is also concerned with efficient implementations. Because the *swap* operation cannot completely replace copying, there will occasionally be need for a *copy* operation. For efficiency, in Ada it is often desirable to provide this operation, as well as a equality comparison operation, even in basic components. Again, Section 6.3 discusses the reasoning behind this recommendation. Also note that this recommendation is made because of efficiency concerns present *when using the Ada language*. Other languages may not impose the same penalties on operations constructed this way.

Guideline 17:

Provide *copy* and *is_equal* operators for all abstractions. Although these are really secondary operations, in the cases where they are needed, the additional costs of all the procedure calls involved in building one using primitive operations is unnecessary.

5.2.3 Table-Driven Programming

Table-driven programming is a common technique of advanced programming which, unfortunately, appears in computer science literature very infrequently. It is useful wherever dynamic, interpretive behavior is an advantage.

A table-driven algorithm is centered around a data structure often called a "dispatch table," "procedure table," or a "state table." The basic idea of operation is simple—input data is used to "index" into the table to determine what routine or code segment is to be executed next. Moreover, the contents of the dispatch table can be modified to add, replace, or remove entries. This makes table-driven algorithms ideal for applications that need data-driven control of execution, particularly execution of user-supplied routines. Because such table-driven systems can be easily changed and extended without modifying their source code, they are ideal for several classes of reusable components.

The most common example of a table-driven system is a programming language interpreter. Often, an interpreter is centered around a "translation table" that indicates how each basic operation in the interpreted language is to be implemented. If the interpretive language allows the user to define his own procedures or functions, these can be added to the basic table or stored in a supplementary table. The program being interpreted is then viewed by the interpreter as an incoming data stream, and this data is used to index into the translation or dispatch table(s). Some programmers consider the use of data to drive the control flow in this way as the essence of interpretation. A good example of such an interpreter is given in [Abelson85a].

Also, many object-oriented (OO) languages use table-driven techniques. As part of the run-time environment, a message dispatch table, or even a set of tables, is built. Then when messages are invoked, a "key" based on the message is used to index into the table(s), and control is dispatched to the appropriate point. The message dispatch tables can either be built into the run-time by the compiler in an static OO environment, or constructed and modified on the fly in a more dynamic environment. A good example of how such mechanisms are used in practice is given in [Cox86a].

Dispatch tables for table-driven systems are often constructed using "procedure variables." Unfortunately, Ada does not provide true procedure variables. This is another failing that inhibits the construction of some forms of reusable components. To show how a simple table-driven system can be useful in an Ada component, as well as how one is implemented, consider extending the GPD *concept*.

```
procedure save(file : in text_io.file_type;
               gpd: in gpd_type);
     -- In effect, a "copy" of GPD is placed on the FILE.

procedure restore(file : in      text_io.file_type;
                  gpd: in out gpd_type);
     -- The GPD parameter of RESTORE is mode in out so that its previous
     -- value can be finalized before the new structure is assigned to it.
```

Figure 23. New operations for *GPD_type*

First, consider adding the operations shown in Figure 23 to the GPD specification. The *save* operation would allow any GPD structure to be written to a file, while *restore* would read such a stored structure back from a file to create an in-memory version. To see how this might affect the use of the package, consider a GPD component being used to implement a tree-structured intermediate representation in a compiler. These new operations will allow the compiler to save intermediate structures into a compilation library as well as reload information about previously compiled units.

Now imagine extending the GPD *concept* so that in addition to the predefined types of *integer* and *boolean*, any user-defined type could be stored in a GPD leaf node. How could this be implemented?

Writing the specification for such a node type is a relatively straightforward task. A *user_defined_node_pkg* should be added along side *empty_node_pkg*, *integer_node_pkg*, and *boolean_node_pkg*. Because this package must be able to work with any user-defined type, it should be a generic. At first guess, such a package would look like Figure 24.

```
...
generic
      type user_defined_data is limited private;
      with procedure initialize(data : in out user_defined_data);
      with procedure finalize(data : in out user_defined_data);
      with procedure swap(left       : in out user_defined_data;
                          right      : in out user_defined_data);
package user_defined_node_pkg is

      procedure new_node(data      : in out user_defined_data;
                         node      : in out gpd_type);
      procedure access_data(node     : in out gpd_type;
                            data     : in out user_defined_data);

end user_defined_node_pkg;
...
```

Figure 24. Generic for a User-Defined GPD Node

Figure 24 is not quite complete, however. In order for the *save* and *restore* operations to be able to do their jobs, the user must also provide *write* and *read* routines for his user-defined type. This results in Figure 25. Note that Figure 25 is as simple as possible for this example, but in general, a slightly different interface for save/restore procedures may be needed, as described in Section 9.5.

Unfortunately, actually implementing the *save* and *restore* operations in Ada is much more difficult. In another language, such as C, the programmer could simply use an untyped pointer to store data of a user-defined type, tagging it with some form of "type" tag so that the appropriate *read* and *write* operations are called when the time comes. Then, as users "register" new user-defined types, pointers to the *read* and *write* routines for each new type are added to two separate tables. *Save* and *restore* would then use the "type" tags stored with the data to call the correct user-supplied routines in order to accomplish the desired mission[14].

```
...
generic
        type user_defined_data is limited private;
        with procedure initialize(data : in out user_defined_data);
        with procedure finalize(data : in out user_defined_data);
        with procedure swap(left          : in out user_defined_data;
                                        right       : in out user_defined_data);
        with procedure read(file        : in        text_io.file_type;
                                       data       : in out user_defined_data);
                -- The DATA parameter of READ is mode in out because READ
                -- FINALIZEs the incoming value of DATA before placing the
                -- result of the READ operation in it.
        with procedure write(file        : in text_io.file_type;
                                        data       : in user_defined_data);
                -- In effect, WRITE sends a "copy" of DATA to FILE.
package user_defined_node_pkg is
...
```

Figure 25. Complete Set of Generic Parameters for the *User_Defined_Node_Pkg*

In Ada, the lack of procedure variables frustrates this approach. Many experienced Ada programmers will suggest using tasking in place of procedure variables, since pointers to tasks are allowed. While this approach will work, it is often implemented inadequately.

The tasking features of most Ada implementations are significantly less efficient than procedure calls. Because of this, programmers often try to optimize the number of rendezous required to perform a given function. Rearranging the code in order to reduce the inter-task communications requirements can often distort simple designs into very complex ones. A system that might have a very simple conceptual design as a table-driven system can turn into a morass of tasks.

Alternatively, some programmers may sacrifice the dynamic benefits of table-driven programming, using a large **case** statement as a dispatch table. While this will work for relatively static systems, it has none of the dynamic adaptability that a true table-driven has. For example, statically implementing the *read* and *write* dispatch tables in the *save* and *restore* operations on GPDs would restrict the flexibility of the component. There would only be a fixed set of predefined *read* and *write* operations which the user could select from when creating a new kind of user-defined GPD node. To add new operations to the table would require alteration of the source code for the component. Also, with a **case** statement as the dispatch table, table entries cannot be added, changed, or removed at run time.

Rather than using either of these two approaches, the superior alternative of creating a reusable component defining the abstraction of a procedure variable is suggested. Such a

14. The *finalize* operation for *gpd_type* must be able to call the various *finalize* operations on the user-defined types as well. It can be implemented using table-driven techniques as in the same way as the *save* and *restore* operations.

component is illustrated in Figure 26. Given an abstraction such as this one, implementing a table-driven algorithm is very easy. The question then shifts to the package body of the procedure variable component—how can it be implemented to address the above problems?

```
generic
    type arg_type is limited private;
    with procedure initialize(data : in out arg_type);
    with procedure finalize(data : in out arg_type);
    with procedure swap(left        : in out arg_type;
                        right       : in out arg_type);
package Procedure_Variable_Abstraction is

    type procedure_variable is limited private;
    procedure initialize(data : in out procedure_variable);
        -- This initializes a procedure variable to the conceptual
        -- value "NULL." This must be executed for each procedure_variable
        -- declared.
    procedure finalize(data : in out procedure_variable);
        -- This releases all resources associated with a procedure variable.
        -- It must be executed on each procedure_variable before that
        -- variable goes out of its defining scope.
    procedure swap(left         : in out procedure_variable;
                   right        : in out procedure_variable);

    function procedure_variable_is_null(pv : in procedure_variable)
        return boolean;
        -- Return TRUE iff PV has the conceptual value "NULL,"
        -- return FALSE otherwise.
    procedure reset_procedure_variable(pv : in out procedure_variable);
        -- Sets a procedure variable to the conceptual value "NULL."
        -- This routine is most often used to "erase" the value of
        -- a used procedure variable.

    generic
        with procedure P(a : in out arg_type);
        -- P should not access any variables outside itself (either
        -- global variables or variables in surrounding transient
        -- scopes).
    package Procedure_Definer is

        procedure set_procedure_variable_to_P(pv : in out procedure_variable);
            -- Sets PV to conceptually "point to" the procedure P.

    end Procedure_Definer;

    procedure invoke_procedure(pv   : in      procedure_variable;
                               a    : in out arg_type);
        -- If PV has the conceptual value "NULL," no action is taken.
        -- If PV "points to" some procedure P, P is invoked with
        -- A as its argument.

    UNINITIALIZED_PV : exception;
        -- This exception is raised if a variable of type PROCEDURE_VARIABLE
        -- is declared and then passed to an operation before INITIALIZE
        -- has been called on it.
```

```
private
        type pv_block;
        type procedure_variable is access pv_block;
end Procedure_Variable_Abstraction;
```

Figure 26. Procedure Variable ***Concept***

As a first cut, it is possible to use Ada's tasking features to implement the procedure variable abstraction. As mentioned above, many programmers have used tasking for this purpose. Figure 27 shows a Buhr diagram of the tasking interactions used to implement this abstraction. The Ada code for the corresponding package body is provided in Appendix C, Section 7.

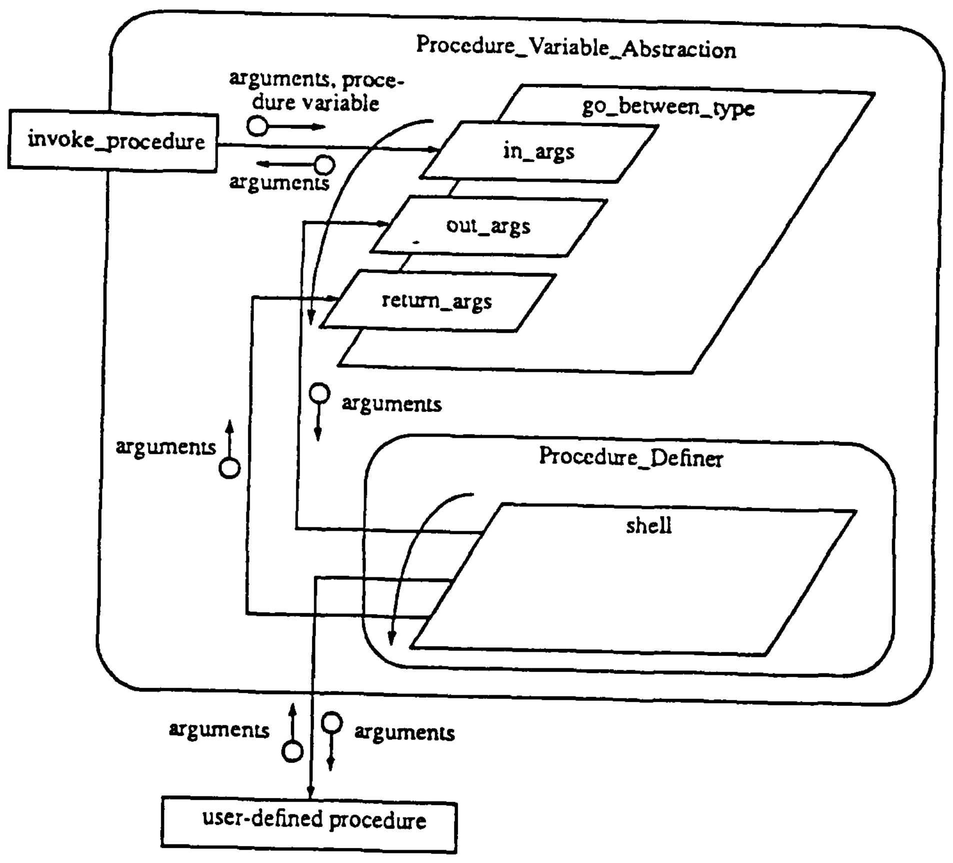


Figure 27. Buhr Diagram of a Tasking Implementation of Procedure Variables

The advantage of encapsulating the tasking implementation of procedure variables is the clean separation it produces between how procedure variables are used in the table-driven system and how procedure variable services are actually provided. This separation ensures that the design of the table-driven system is not concerned with the details of the tasking. The design will be more consistent because there is no opportunity for it to be compromised by the optimization of inter-task communications. The benefits of such a unified and simple conceptual model can be great, especially during maintenance.

Unfortunately, in addition to being the strength of this approach, the encapsulation of implementation details is also its major weakness. The fact that there is no opportunity for optimizing task operations can be a significant drawback, despite the benefits this limitation provides. Some programmers consider the weight of this restriction to be a strong argument against encapsulation in some cases. Fortunately, difficulties such as this can usually be turned into advantages.

In particular, by encapsulating the procedure variable implementation, the possibility of alternate, more efficient implementations is opened. While the package body presented in Section 7 of Appendix C has the advantage of being portable to all machines, it is possible to sacrifice portability to gain efficiency.

The most common way of efficiently implementing procedure variables is to use the predefined *ADDRESS* attribute. When used on a program unit such as a procedure, this attribute returns the address of the first memory storage unit allocated to that program unit. For many common machine architectures, such an address can be turned into an indirect procedure call easily. This can be done either through machine code insertion within Ada, or by using the *INTERFACE* pragma to call out to another language. Both methods are non-portable. Section 8 in Appendix C shows a version of the *Procedure_Variable_Abstraction* package body which interfaces with a C routine to invoke the pointer. Section 9 in Appendix C shows the corresponding C routine.

The difficulty with this implementation is that it fails to account for the "access link" of the procedure held in the procedure variable [Aho86a, page 418]. The term "access link" is used here to refer to the method used by the compiler to allow access within a procedure to variables declared in surrounding scopes, whatever that method might be. Because the surrounding scopes of a nested procedure may only exist temporarily, special care needs to be taken when nested procedures are assigned to procedure variables. If a procedure variable containing a nested procedure is invoked *after* its enclosing scope has been exited, nonlocal references made by that procedure may be invalid.

In order to address this problem, the specification in Figure 26 was written using a simple solution—procedures assigned to procedure variables shall not access nonlocal variables. This restriction is not the only solution to the access link problem, but it is the easiest to implement when creating a procedure variable abstraction as a new user-defined type in Ada. By adding this constraint, the optimization of Section 8 of Appendix C is likely to work without changes for many more compilers. Unfortunately, this restriction limits some of the usefulness of procedure variables. However, this restriction can be relaxed, if necessary.

For cases where a shared set of nonlocal variables between procedures and procedure variables is needed, an extended solution is possible. By replacing the specification of the *Procedure_Definer* subpackage with the version shown in Figure 28, the *Procedure_Variable_Abstraction* can be extended to allow the user to specify the type of "environment" each procedure variable executes within. The user of the abstraction can then place nonlocal values accessed by procedure variables in an "environment." Procedure variables, as well as other user-defined routines, can share access to a single environment, or have independent environments for nonlocals. For the implementation of this unit, the type *pv_block* can be extended to contain a pointer to the environmental data for each procedure variable.

```
  . . .
        generic
            type environment_type is limited private;
            with procedure initialize(data : in out environment_type);
            with procedure finalize(data : in out environment_type);
            with procedure swap(left       : in out environment_type;
                                right      : in out environment_type);

            with procedure p(a   : in out arg_type;
                             e   : in out environment_type);
        package Procedure_Definer is

            procedure set_procedure_variable_to_P(
                pv    : in out procedure_variable;
                e     : in out environment_type);
                -- Sets PV to conceptually "point to" the procedure P.
                -- Also takes the environment E and puts it in PV (upon
                -- completion, E has the value given by "initialize").
                -- Whenever PV is invoked, P will be called with E as
                -- one of its parameters.

            procedure access_procedure_environment(
                pv    : in out procedure_variable;
                e     : in out environment_type);
                -- "Swaps" E with the environment stored in PV.

        end Procedure_Definer;
  . . .
```

Figure 28. Modified *Procedure_Definer* which allows Shared "Environments"

Fortunately, encapsulation of the abstraction allows *both* versions of the package to be provided. An optimized version of the procedure variable abstraction can be used on machines where one is available, and the completely portable version is ready as a backup for machines where an optimized version has not yet been written. This addresses both the efficiency and portability problems of table-driven systems, which are so useful in constructing flexible, reusable components.

Because procedure variables are so commonly used, there have been many recommendations for them to be included in the Ada 9X revision process. This encapsulated procedure abstraction is also upwardly compatible with any 9X changes. If procedure variables are supported in the new language revision, only the body of the *Procedure_Variable_Abstraction* package needs to be rewritten, providing an implementation which is both efficient *and* portable. In addition, if the abstraction were supported at the compiler level, a less inhibiting solution to the access link problem could be provided.

5.3 SUPPORT FOR *CONTEXT*—PARAMETERIZATION MANAGEMENT

Failure to provide complete encapsulation, a facility which is necessary to effectively separate *context* from the *concept*, has already been addressed in Section 5.1. The second class of features which support *context* are "parameterization management" features. These features are language mechanisms that allow a person to deal with a very large *context*—i.e, a large number of parameters. These features are named after the parameterization management problem, originally introduced in Section 2.6.

It is often stated that the biggest potential benefits from reuse come from reapplying relatively large components [Biggerstaff87a]. Intuitively, this idea makes sense. It is cheaper to reuse a whole subsystem consisting of 10 subcomponents than to reuse 10 smaller individual components and integrate them together yourself.

Unfortunately, as a component grows in size, the amount of functionality it encompasses also grows. Likewise, the number of choices that should be left up to the reuser so he can tailor the component also grows. In other words, larger components have more parameters, using whatever parameterization mechanism(s) the language of concern supports[15].

Also unfortunately, as a component grows in size, and as the number of "layers"[16] that are used to define that component grow, the size of the *context*—the number of parameters—grows more than linearly. In fact, it grows exponentially. This is not surprising, considering the rates at which other factors increase with code size.

This growth rate also results from the fact that larger components are often constructed of smaller components. Certainly, the larger component will choose some parameter values for such a subcomponent. However, the larger component should only choose values for the parameters that are determined by the way in which this subcomponent is being used. In order to be a flexible as possible, parameters that are not determined by the structure of the larger component should be deferred to the reuser. Thus, the "unbound" parameters of the subcomponent are included as part of the *context* of the larger component.

15. See Section 2.3 for a description of parameterization mechanisms.

16. See Figure 4, which depicts a layered system of components.

This indicates that at some point, the size of the *context* will grow large enough that an average person cannot effectively deal with it. In fact, Miller's ''magical number seven, plus or minus two,'' [Miller56a] indicates that it will not take long for a component to reach the point where it is difficult for a reuser to tailor such a component. Miller's work indicates that the number of separate pieces of information a person can hold in short term memory is very limited.

> Short-term memory imposes one of our greatest limitations in building large software systems, because it prohibits us from being able to consciously attend to and manipulate all of the interacting parts of a complex system at once. [Curtis89a, p. 270]

Fortunately, this limit to human capability can easily be overcome by *chunking* related pieces of information into a new, larger unit. Unfortunately, however, an unstructured group of parameters can be difficult to chunk effectively. Further, the fact that *context* size may grow as much as exponentially indicates that the point where the size will be difficult to deal with will be reached while components are relatively small.

To see how this realizes itself in Ada, consider a component that has seven generic formal types, each of which represents a different abstract data type. In the generic formal part of the Ada package specification, there would have to be generic parameters for each of the types, and all of the basic operations on those types (*initialize*, *finalize*, *swap*, *copy*, and *is_equal*). This alone could result in as many as 42 generic parameters! But this is still a small component by comparison, because no mention has been made of any subcomponents it uses, or the parameters that might be added to the larger component to tailor these subcomponents. In addition, these parameters have a natural organization that leads to chunking. But the shear size and complexity the reuser is presented with on reading the specification has an impact that can be intuitively felt.

The parameterization management problem, then, is the problem of finding mechanisms by which reusers and designers alike can cope with the expanding *contexts* of medium and large components. Unfortunately, and despite section 2.3, there is no commonly accepted model of what parameterization (i.e., tailoring) actually is, or how it should be used. Thus, although several existing language mechanisms have been proposed as potential solutions to the parameterization management problem, there is no way of knowing how much, if any, of the problem they address.

Although this paper does not offer a complete solution to the parameterization management problem, some potential parameterization management mechanisms can be presented. Although there is little more than intuitive evidence that these mechanisms will solve the problem, they certainly offer an improvement over Ada's current situation. Proposed solutions include:

a. Inheritance mechanisms

b. Modularly structured parameters, as in [Goguen84a] and [Tracz90c]

c. Partial instantiation of generics

d. Package types, which can be passed as generic formal parameters

It is also worth noting that the majority of these approaches suggest trying to impose a structure on the parameters so that they can be naturally chunked without significant effort.

Although the GPD concept presented in Section 3, because of its simplicity, does not illustrate how such features would be used, it is clear that Ada does not provide them. Instead, generic parameters in Ada exist and are declared using a flat, unstructured approach, exacerbating the problem. Several recommendations have been made to include some of these mechanisms, particularly partial instantiation or package types in the Ada 9X revision to rectify this problem, but no underlying model of parameterization has been proposed.

Because there is no work-around available within the confines of the Ada language itself, the parameterization management problem can force component designers to choose less reusable designs. As Guideline 3, presented in Section 4.3, states, the most desirable case is reached when all of the parameters to a component's *concept* are represented as generic parameters. However, the component grows more difficult to understand and use as the number of these unstructured parameters grows.

As a result, the designer may reach a point of diminishing returns where *making the component more flexible makes it overly difficult to use*. Component writers must be aware of this problem, and must carefully temper adherence to Guideline 3 to ensure that the potential for reusing the resulting component is acceptable.

It should also be clear that the conditions under which this point of diminishing returns is reached are dependent on the tailoring mechanism being used. The mechanism being used is in turn dependent on the language being used. In this case, Ada, and Ada's generic parameterization mechanism, are being used. Different languages or tailoring mechanisms may influence how soon the point of diminishing returns is reached. Also, this perspective allows one to view the alternative solutions presented previously as providing new tailoring mechanisms that push this point farther away, so that programmers can deal effectively with larger and larger amounts of *context*. For now, however, the component writer should be aware of this problem and how it can cause exceptionally general components to be less reusable.

6. AREAS WHERE DESIGN DECISIONS COMMONLY RESTRICT REUSABILITY

Section 5 describes some of the subtle concerns that arise when designing reusable components in Ada, and that are based on the peculiarities of that programming language. However, there are additional concerns that arise not from inherent limitations of the language itself, but rather from design or implementation decisions that were made without considering their impact on reusability.

When a software engineer or programmer creates a new module, he often has a natural tendency to optimize the design of the module to the requirements as he sees them. He may even use very specific information about the problem the module will be solving to drive not only the algorithm selection, but also the design of the module's interface. Knowledge about other modules that will work in conjunction with the one under consideration may also influence the design of the interface. While this skill is very valuable, particularly in the design of modules for embedded or real-time systems, it can lead to less reusable components.

Because no software components industry currently exists, it is reasonable to believe reusable components that will be written in the near future will be written as part of a specific software project. It is also likely that the design of these components will be driven *first* by the needs of a specific application, and driven by reusability and generality concerns *second*. While this approach may be sound economically, it is possible that it may lead to implicit assumptions, for example, about how the module interacts with the remainder of the application, that are designed or written into the component. This is most likely not a failing of the engineer or programmer, but rather a result of the single-application-oriented requirements, which did not describe a generalized component.

In order to assist such component designers in the process of generalizing application-specific components into more general ones, this section presents several common failings of components that were intended to be reusable. The common concerns highlighted here are memory management approaches, concurrency protection models, iterators for abstract data types, and save and restore operations for abstract data types. By presenting these concerns to the component designer, it is possible to bring out the various design decisions that are often very important for reusability, but which may be made implicitly, without thought, if they do not appear in the application-oriented requirements for a module.

The following subsections describe the most common areas where component writers fail to consider the impact on reusability. These sections are based loosely on Booch's component taxonomy [Booch87a], which very roughly categorizes components based on how they address each concern.

6.1 MEMORY MANAGEMENT

Management of dynamically allocated memory is perhaps one of the most well known problems when dealing with reusable components, because it is such a common part of data structure implementations in general. In the case of a reusable component that exports an abstract data type, memory management involves the method of allocating space for new objects or objects that are growing, and reclaiming space from objects that are no longer needed or that are shrinking. For the purposes of this section, consider the requirements that a particular memory management scheme places on the component user. What responsibility does the user have in the memory management scheme, and what portions of the scheme are visible to the user?

While there are many approaches to memory management, a large number of all implementations fall under one of ten basic "models," from the component user's point of view. These models are differentiated depending on how they deal with four basic questions:

a. Is memory space reclamation handled by the component, or left up to the language run-time system or operating system?

b. Does the user actively participate in the memory management process?

c. Is the memory management portion of the component protected against use by concurrently executing threads of control?

d. Are there limits on the total amount of memory that can be used by the component?

For each of these questions, there are apparently two common answers, and the combinations of answers result in the ten "mainstream" memory management models[17]. While there are many other alternatives, including hybrid schemes, these models seem to be the most prevalent among dynamic data structure modules. The common facets of these memory management models are:

a. Is memory space reclamation handled by the component, or left up to the language run-time system or operating system?

    (1) **Unmanaged** components provide no mechanism for space reclamation, usually relying on the underlying run-time system to provide automatic garbage collection.

17. Not all combinations of answers result in useful memory management schemes, for example, *explicitly unmanaged*.

(2) **Managed** components do provide a mechanism for space reclamation.

b. Does the user actively participate in the memory management process? (This question only applies to **managed** components.)

(1) **Explicitly** managed components require the user to call some form of "free" or "deallocate" routine in order to explicitly reclaim space.

(2) **Transparently** managed components do not require any explicit action on the user's part to "free" unneeded memory space. Instead, they usually implement some local form of garbage collection within the component. Such components often rely on the user to consistently call the appropriate *finalize* operations on variables when they are leaving scope.

c. Is the memory management portion of the component protected against use by concurrently executing threads of control? (Again, this question only applies to **managed** components.)

(1) **Controlled** components are built so that exported *free* or *finalize* routines work correctly even if simultaneously executed by multiple threads of control.

(2) **Uncontrolled** components require the corresponding *free* and *finalize* routines to only be executed by one thread of control at a time.

d. Are there limits on the total amount of memory that can be used by the component? (This question applies to both **managed** and **unmanaged** components.)

(1) **Limited** components have an upper bound on how much memory can be consumed by all of the active objects of the exported abstract data type at one time. For example, a linked list abstraction may allow the reuser to set an upper limit on the total amount of memory that may be used by all of the lists that are currently allocated.

(2) **Unlimited** components do not allow the reuser to place a bound on their memory consumption.

While there are many other alternatives available, these memory management categories are the most commonly used. The disadvantage of using another memory management technique in a reusable component is that such an approach will make the component more difficult to understand and use, from the user's perspective. There is certainly such a place for such alternative schemes, but component designers should be aware of the commonly understood approaches, as well as the potential effect on reuse when deviating from them.

In addition, the labeling of reusable components with their memory management approach would be helpful. Not only would it be a small step toward making the components more understandable, but it would also help greatly in the component selection process. For example, a reuser trying to select a component for use in an embedded system could easily see whether it was designed to be **limited** or not. A preliminary pass at a labeling scheme based on this concept is presented in Appendix A.

6.2 CONCURRENCY

Another concern component designers have is that of "hardening" components against concurrent access. This concern arises from the fact that Ada provides constructs to explicitly describe concurrent threads of execution within a single program. As a result, component designers must explicitly address the problem of more than one Ada task simultaneously accessing the operations within a component, or simultaneously accessing the same object of some abstract data type exported by the component.

The tradeoff between protection and efficiency is often at the heart of discussions about protecting components from concurrent access. Clearly, a component which does not address concurrency protection is less reusable, since it will be difficult to employ in a program that relies on tasking. However, the inefficiency of many concurrency approaches may make a concurrency hardened component less desirable in more conventional programs.

This conflict has led to quite a few concurrency protection approaches. Component designers, trying hard to provide maximal protection with minimal efficiency impact, often create "custom made" protection schemes for new components. Unfortunately, exotic protection schemes often have a steep learning curve. Such schemes can also offer traps to the unwary, leading to hidden race conditions or protection failures which are hard to track down and eliminate.

Because of the variety of methods that are currently used in concurrency protection for Ada components, no complete taxonomy is provided here. Instead, a very rough categorization is provided, based on Booch's work [Booch87a, pp. 41-42]:

a. **Sequential** components provide no concurrency protection whatsoever.

b. **Shared** components provide concurrency protection for the internal state, both hidden and visible portions, of the package (or instantiation) exporting the abstraction (for example, internal hash tables, caches, etc.). **Shared** components do not, however, provide any built in protection for objects of the exported abstract type(s). Multiple threads of control can "share" the facility, although they must provide some other protection mechanism of their own to share objects from that facility.

c. **Guarded** components ensure that each object of an exported abstract type has some form of abstract "lock," but multiple threads accessing a shared object must obey the associated "locking" conventions. In other words, the necessary framework for mutual exclusion at the object level is provided, but it is up to the client tasks to assure that mutual exclusion is maintained by obeying the conventions of this framework. Note that **guarded** components also provide the component state protections of **shared** components.

d. **Concurrent** components guarantee that each object of an exported abstract type may only be accessed by one thread at a time. The component assumes responsibility for ensuring mutual exclusion, regardless of the actions of the client tasks. This makes the framework of mechanisms for ensuring mutual exclusion transparent to the user (client). Note that **concurrent** components also provide the component state protections of **shared** components.

e. **Multiple** components have the same responsibility as **concurrent** components, ensuring that objects of exported types and the component itself maintain consistent states in the face of arbitrary, concurrent accesses. The difference is that **multiple** components optimize accesses to a single object (or to the shared internal state of a component) for maximal concurrency, for example, allowing multiple, simultaneous read operations but only allowing exclusive write operations.

While these alternatives only define the broad boundaries of available concurrency protection models, they do illustrate the basic concepts from which more exotic approaches are often built. As with different memory management models, the most important point is that simpler, more well-known protection models are easier for the reuser to understand, and also easier for the reuser to correctly apply.

When designing a new component that will include concurrency protection, the designer should very carefully choose the protection model he will use. A knowledge of approaches commonly used in the past will help in choosing a model that will be more easily understood and applied.

However, if possible, the concurrency protection model should *not* be directly visible in the *concept*. In general, concurrency protection is a property of the implementation, not of the abstraction itself. The primary reason for concurrency protection models that are visible in the concept—**guarded** components, for example—is to allow the client an opportunity to optimize the protection operations. The reuser has more knowledge about the intended use of a component, and can therefore somewhat compensate for costly synchronizing steps. Unfortunately, exposing the protection model through the component's specification makes it more difficult to change that model. It also places responsibility for maintaining the model on the reuser's shoulders, instead of inside the component where it should be. Unnecessarily tying the concurrency protection model to the abstract functionality of the component in this way reduces reusability and maintainability.

There are certainly many cases where the limitations of Ada, or the requirements of a particular project, will require alternative protection schemes, however. As concurrency hardened components become more common, and widely accepted concurrency protection models arise, this grouping should be revised and expanded.

6.3 ITERATORS

The concept of "looping" or "iterating over" a group of items is fundamental to computer science. However, one of the most difficult problems in designing a reusable component is designing constructs to perform such operations. Reusable components that export abstract data types often supply operations to iterate over the parts of an object of that type. Other components that have visible state—for example, a component that embodies a parser, and that has a visible state stack—may provide some form of iteration over that visible state. Unfortunately, constructing general purpose iterators in strongly-encapsulated components is often very difficult in practice.

To give an example of how iterator construction can be deceptively simple, consider an iterator for the *gpd_type* discussed throughout this paper. A single GPD node can be viewed as the root of a tree (or directed graph) structure. An iterator would then "walk," or traverse, this tree, calling some user-defined operation on each node along the way. For this example, consider an iterator which provides an "inorder" traversal of the GPD structure rooted at a given node, assuring that each node will only be visited once. Such an operation, described in Ada, might appear as in Figure 29.

```
generic
    with procedure user_defined_operation(n : in gpd_type);
procedure in_order_traversal(n : in gpd_type);
```

Figure 29. Declaration of a GPD Iterator

While this iterator is very simple, and using it is straightforward, it is very limited in functionality. It allows the user to *inspect* the contents of the GPD structure in a predefined order, without *altering* either the structure or the contents of the GPD nodes. It also requires that *every* node in the structure be visited, but each can only be visited once.

If the iterator in Figure 29 is compared with the built-in facilities Ada provides for iterating over arrays, many of the limitations become clear. Using **for**, **while**, or **loop** statements in Ada, a programmer can easily construct iterations that can visit the elements of an array in almost any order, change the contents of any element in the array (without affecting the *structure* of the array itself), and terminate under almost any circumstance. Yet for a strongly encapsulated—e.g., **limited private**—type, the only facilities the user may have for iteration are those exported through the component's specification.

There two main ways of addressing this problem. First, it is possible to define a component so that exportation of an explicit iteration construct is unnecessary. Second, one can attempt to export a comprehensive iteration construct to cover the required looping functionality. Further, if the choice to provide an iterator is made, there are two possibilities: a *passive* iterator, or an *active* iterator. Each of these approaches will be discussed in turn.

6.3.1 Eliminating the Need for an Explicit Iterator

To see how the need for an explicitly exported iterator can be eliminated, recall the concept of *primary* and *secondary* operations introduced in section 5.2.2.3 [Weide86a]. Primary operations are the "primitives" available for manipulating a given abstract data type. The set of primary operations should be complete enough for the reuser to do everything needed to an object of the abstract type. Secondary operations, by definition, can be constructed from the primary operations and are thus not necessary in a component. In Ada, the most common iteration constructs programmers work with are built from the primary operations available on arrays and on discrete types (array index types).

Iterators are actually secondary operations. If a component were to export a "complete" set of primary operations, the programmer could easily build his own iterators from them. "Complete," in this context, means primitives that are sufficient to access and change every subpart within an object of an abstract type. In a sequence, the subparts would be the individual elements of the sequence, while in a graph structure, the individual elements would be the nodes within the graph.

This approach to iterator construction is not only simple, it results in elegant and easy to use components. It greatly simplifies the programmer's task when constructing iterators, since the programmer can use the looping features of the language without difficulty and to full effect.

However, some component designers wish to limit the set of primary operations in a *concept* in order to allow more freedom in the implementation. For example a hash table may describe a mapping from some input space (the tokens) to some output space (the table entries), but the primary operations that are provided may not allow construction of an iterator. Consider the *concept* shown in Figure 30, which defines a generalization of such a hash table—a unidirectional associative memory.

18. This component is derived from the "Almost Constant Map Facility" presented in [Weide86b]. It has been converted to Ada and slightly modified for this example.

```
generic
        type Domain_Type is limited private; -- the domain of the associative map
        with procedure Swap(left, right : in out Domain_Type);
        with procedure Initialize(d : in out Domain_Type);
        with procedure Finalize(d : in out Domain_Type);
        with procedure Copy(from      : in      Domain_Type;
                            into      : in out Domain_Type);
        with function Is_Equal(left, right : in Domain_Type) return boolean;

        type Range_Type is limited private; -- the Range of the associative map
        with procedure Swap(left, right : in out Range_Type);
        with procedure Initialize(r : in out Range_Type);
        with procedure Finalize(r : in out Range_Type);
        with procedure Copy(from      : in      Range_Type;
                            into      : in out Range_Type);
        with function Is_Equal(left, right : in Range_Type) return boolean;

package Unidirectional_Associative_Memory_Concept is

        Type UAM_map is limited private;
        -- basic operations defined for every type
        procedure Swap(left, right : in out UAM_map);
        procedure Initialize(m : in out UAM_map);
        procedure Finalize(m : in out UAM_map);
        procedure Copy(from      : in      UAM_map;
                       into      : in out UAM_map);
        function Is_Equal(left, right : in UAM_map) return boolean;

        -- primary operations for UAM_maps
        procedure get_default(m  : in      UAM_map;
                              r  : in out Range_Type);
                -- The value of R when the call is made is finalized; then a copy
                -- of the default value of M is placed in R.  M is not affected.

        function is_constant(m : in UAM_map) return boolean;
                -- Returns false if there exists a D in Domain_Type such that
                -- M(D) /= GET_DEFAULT(M).

        function is_not_default(m  : in UAM_map;
                                d  : in Domain_Type) return boolean;
                -- Returns true iff M(D) /= GET_DEFAULT(M) (i.e., if D has been
                -- entered into the map M).

        procedure reset(m  : in out UAM_map;
                        r  : in out Range_Type);
                -- finalizes all the old values in M, and resets it so that R is
                -- the new default value for M.  The original default value for
                -- M is also finalized, and on exit R has been "consumed" and
                -- contains the value of a newly initialized Range_Type variable.

        procedure access(m  : in out UAM_map;
                         d  : in      Domain_Type;
                         r  : in out Range_Type);
                -- The map M is applied to the value D (D is used to "index" into
                -- M).  The resulting Range_Type value is "swapped" with the value of
                -- R.  On exit, R contains the old value of M(D), and M(D) contains
        -- the old value of R.

end Unidirectional_Associative_Memory_Concept;
```

Figure 30. Unidirectional Associative Memory *Concept*[18]

The operation *access* allows any entry to be examined or altered, and together with the other operations allow any arbitrary associative mapping to be constructed. Unfortunately, there is no given ordering enforced on the input domain *D*, and no way to find out which values in the domain have been given special values (i.e., which tokens have been entered into the table) short of calling *is_not_default* for every value in the domain. Also, notice that the domain *D* need not even be finite. Because of these properties, there is no feasible way for the user of such a component to iterate over the set of "interesting" entries.

As a result, the component author may wish to export some form of iterator construct in order to supply this functionality. In fact, the component author must be able to perform this kind of operation inside the component in order to call the appropriate *finalize* operations on all of the tokens and values that have been inserted into the table, and also to implement the *copy* and *is_equal* operations for *UAM_map*s. But how can this capability be presented to the user in an effective way?

It is still possible to alter the interface of the Unidirectional Associative Memory component so that enough primary operations are provided for the user to construct an iterator. In [Weide86b], this component includes the additional primary operation shown in Figure 31. The *remove_entry* operation allows all entries in the associative memory to be visited in some implementation defined order, removing each entry from the memory map as it is visited. This capability, which is destructive to the original map, allows arbitrary iterators to be constructed without requiring data to be copied from the original memory map. While it is certainly less intuitive to use than a **for** loop, or the iterator shown in Figure 29, it provides all necessary functionality.

6.3.2 Passive Iterators

Alternatively, the component designer may choose to export an iterator construct he feels is easier to utilize. Booch classifies such constructs into *passive* and *active*[19] iterators [Booch87a, pp. 157-161]. In a passive iterator, only a single operation that encapsulates the operation of the iterator is exported, as in Figure 29. An active iterator, on the other hand, has its internal method of operation somewhat exposed through a collection of exported operations. The reuser can use these operations, along with Ada's looping constructs, to build a code fragment that performs the iteration of his choice.

19. In [Bishop90a], these are referred to as *iterators* and *generators*, respectively.

```
procedure remove_entry(m  : in out UAM_map;
                       d  : in out Domain_Type;
                       r  : in out Range_Type);
    -- This function requires M to have at least one entry (i.e.,
    -- IS_CONSTANT(M) = false), and raises UAM_ERROR otherwise.
    -- This values of D and R are finalized. If M is not constant, then
    -- D is given the value of one value in the domain such that
    -- M(D) /= GET_DEFAULT(M). R is given the value of M(D), then
    -- M is altered so that M(D) now has the value GET_DEFAULT(M) (i.e.,
    -- the entry D, and the associated value R, are removed from M).

UAM_error : exception;
```

Figure 31. An Additional Primary Operation for the Unidirectional Associative Memory *Concept*

The difficulty of designing either form of iterator is constructing one that does not have the drawbacks listed previously for the iterator in Figure 29. These drawbacks can be highlighted by asking the following questions about a possible iterator design:

a. Can the user perform a *destructive* iteration, where the subparts of the object, but not the structure of the object itself, are altered?

b. Can the user control the order of visitation?

c. Is the user forced to visit every subpart of the object, or can he specify when the iteration should terminate?

d. Can the user visit some or all of the subparts more than once?

The problem of designing iterators for encapsulated types is discussed in more detail in [Bishop90a].

Attempting to provide a functionally complete passive iterator often results in an unwieldy, difficult to understand operation, as shown in Figure 32. In addition, the iterators shown in Figure 32 do not provide a complete base for any iterator the user might desire. The reuser's needs for iterating, the designer's need for hiding detail, and the difficulty of usage must all be weighed when creating such an iterator. The biggest advantage of passive iterators is their encapsulation—the reuser cannot affect the *structure* of the object during the iteration, he may only alter its contents.

```
        -- An iterator for GPD_type.
generic
        type UD_parameters is limited private;
                -- A user-defined type used to pass parameters to the
                -- user-defined operation. Successive invocations of the
                -- user-defined operation can share information by saving
                -- it in their PARMS as well.
        with procedure user_defined_operation(
                node            : in out gpd_type;
                parms           : in out UD_parameters;
                continue        :     out boolean
                        -- If FALSE is returned here, the iteration is prematurely
                        -- terminated without calling this operation for any of
                        -- the remaining nodes in the GPD structure ROOT).
        );
procedure in_order_traversal(
        root    : in out gpd_type;
        parms   : in out UD_parameters
        );


        -- An iterator for the Unidirectional Associative Memory Concept.
generic
        type UD_parameters is limited private;
                -- A user-defined type used to pass parameters to the
                -- user-defined operation. Successive invocations of the
                -- user-defined operation can share information by saving
                -- it in their PARMS as well.
        with procedure user_defined_operation(
                d               : in     Domain_Type;
                r               : in out Range_Type;
                parms           : in out UD_parameters;
                remove_it       :     out boolean;
                        -- If this is true, the entry for D in M will be removed
                        -- when this procedure returns (and before it is called
                        -- again for the next entry in M).
                continue :          out boolean
                        -- If FALSE is returned here, the iteration is prematurely
                        -- terminated without calling this operation for any of
                        -- the remaining entries in M).
        );
procedure iterate_over_associative_map(
        m       : in out UAM_map;
        parms   : in out UD_parameters
   );
```

Figure 32. An Attempt at a Two Complete Passive Iterators

6.3.3 Active Iterators

Active iterators offer the potential of more complete functionality, at the expense of difficulty of usage and understanding. Often, active iterators involve the creation of a distinct object that contains the current state of the iteration (which object is being iterated over, which subpart of that object is next, etc.) [Booch87a, p. 157-159]. An example of a simple active iterator, corresponding to the passive iterator of Figure 29, is shown in Figure 33.

```
type in_order_traversal is limited private;
-- declarations of basic operations for this type omitted for
-- simplicity
procedure initialize(i       : in out in_order_traversal;
                     over    : in     gpd_type);
      -- initializes I for an iteration over the GPD structure OVER, setting
      -- the "current location" of I to be the "first" node in that
      -- structure (as defined by the ordering imposed by IN_ORDER_TRAVERSAL).

procedure advance(i : in out in_order_traversal);
      -- Causes I to advance its "current location" to the "next"
      -- node in the GPD structure I is iterating over (where "next"
      -- is defined by the ordering imposed by IN_ORDER_TRAVERSAL). If
      -- has_completed(i) = true, or if I is not initialized,
      -- ITERATOR_ERROR is raised.

procedure value_of(i              : in     in_order_traversal;
                   next_node      : in out gpd_type);
      -- Inspects I, and returns the node at the "current location"
      -- of I in NEXT_NODE (NEXT_NODE is finalized first, to erase
      -- any previous value). If has_completed(I) = true, or if I
      -- has not been initialized, then ITERATOR_ERROR is raised.

function has_completed(i : in in_order_traversal) return boolean;
      -- returns true iff every node in the GPD structure has already
      -- been visited by calling ADVANCE(I). Raises ITERATOR_ERROR if
      -- I has not been initialized.

procedure finalize(i : in out in_order_traversal);
      -- Terminates the iteration corresponding to I, freeing up
      -- all resources it consumed.

iterator_error : exception;
```

Figure 33. A Simple Active Iterator

A significant difficulty with active iterators, however, is that they open up the possibility that the object being iterated over (the *UAM_map*, for example) may be *structurally altered* during the execution of the iterator. "In the manner we have defined it, the iterator gives us great flexibility in composing an iterator, but it is relatively unprotected. Hence, there is the potential for clients to abuse this abstraction" [Booch87a, p. 159].

Fc:tunately, it is possible to add the necessary protection to an active iterator. For example, Figure 34 shows an active iterator for the Unidirectional Associative Memory *concept*. Notice that rather than create separate objects to hold the state of the iterator, this state is folded into the object itself. This allows other operations to ensure that the object is not inside an iteration before they change its structure.

```
procedure start_iteration(m : in out UAM_map);
        -- places the map M into a state so that it can be iterated over
        -- as if it were an array. A subsequent call to end_iteration
        -- will terminate the iteration over M. While M is being iterated
        -- over, destructive routines (like Reset and Access) will cause
        -- OPERATION_NOT_ALLOWED to be raised. Nested "blocks" of
        -- start_iteration and end_iteration calls are allowed.

function being_iterated_over(m : in UAM_map) return boolean;
        -- returns true iff start_iteration has been called on M (signifying
        -- the beginning of an iteration), but no corresponding
        -- end_iteration has been called (to signify the end of that
        -- iterator).

function size_of(m : in UAM_map) return integer;
        -- returns the number of entries in M.

procedure access_nth_entry(m  : in out UAM_map;
                           N  : in     integer;
                           d  : in out Domain_Type;
                           r  : in out Range_Type);
        -- This procedure can only be called when being_iteraed_over(m) = true,
        -- and it will raise OPERATION_NOT_ALLOWED otherwise. When M is being
        -- iterated over, it can be considered to be an array from 1 to
        -- SIZE_OF(M) entries, each of which corresponds to a (Domain, Range)
        -- pair.
        --
        -- When this routine is called, the current value of D is finalized.
        -- then, the Nth pair in M is accessed. The Domain value of this
        -- pair is copied into D, and the Range value of this pair is "swapped"
        -- with the current contents of R. This routine can later be called
        -- to place this original Range value back into M.
        --
        -- Access_Nth_Entry can be called as many times as desired, and there
        -- is no restriction on which elements of the conceptual array are
        -- accessed, or in what order.

procedure end_iteration(m : in out UAM_map);
        -- stops the current iteration over M, allowing destructive operations
        -- to be used once again. ITERATOR_ERROR is raised if
        -- BEING_ITERATED_OVER(m) = false.


iterator_error : exception;
operation_not_allowed : exception;
```

Figure 34. An Active, Random Access Iterator

Folding the iterator state into the object in this manner is not required for protection. It is certainly possible to create separate objects for maintaining iterator state (as in Figure 33). It is also possible to split the iterator state, placing some of it on the object being iterated over, and placing the rest inside a separate object. The central point is that destructive operations must be able to discern whether a given object is currently being iterated over, irrespective of the implementation method chosen for maintaining iterator state information.

Also, it is important for the active iterator implementation to support multiple iterations over the same object (as shown in Figure 34). Otherwise, the reuser would not be able to nest iterations within other iterations, a common algorithmic technique. This requirement raises additional concerns about where iterator state can be stored.

The iterator state (for all currently active iterations) must be preserved both during nested iterations over the same object and, for components that support concurrency, during concurrent iterations over the same object. Some active iterators do not require independent iterator state information for each nested or concurrently active iteration. Others do require independent state for each iteration, and explicit iterator state objects are the only way to ensure that calls to the iteration routines (for example, *value_of* or *has_completed* in Figure 33) are matched with the correct iterator state. The placement of iterator state information will be discussed again in Section 6.3.4.

Figure 34 also shows another technique for designing effective active iterators. In this example, the active iterator allows the reuser to treat the associative map being iterated over *as if it were an array*[20]. Working inside a "block" of code delimited by calls to *start_iteration* and *end_iteration*, the reuser can use a **for** loop, or any other looping construct, to create his own iteration. As Booch noted, active iterators can certainly provide the reuser with a great deal of power.

This form of active iterator could be termed a *random access* active iterator. It associates the idea of an array *index* with the iterator, an integer in this case. The type used for the index could certainly be turned into a generic parameter, but that was not done here for simplicity of the example. Using such an index, the reuser can access any element of the map, in any order he chooses, simply by manipulating the index he requests. Also, since Ada's **for** looping statement is a natural means of applying the parts of an active, random access iterator, such iterators might be termed *for loop* iterators.

By allowing the reuser to view the *UAM_map* as an array in his mind, active, random access iterators significantly increase understandability and usability. A similar active iterator construct, a *sequential access* iterator, offers similar benefits.

Sequential access iterators are often used in **while** loops, rather than **for** loops. Thus, they may also be referred to as *while loop* iterators. The iterator presented previously in Figure 33 is an example of an active, sequential access iterator.

20. Wheeler, David A. 1990. Institute for Defense Analyses, private communication.

6.3.4 Recommendations on Iterator Construction

There are many possibilities when choosing an iterator mechanism for a reusable component. It is possible to create components that do not need iterators, as shown in Section 6.3.1, although this may force the reuser to employ a different strategy for building his iteration. Passive iterators can be provided, and are easy to define, although they may not meet all the reuser's needs. Both sequential and random access iterators provide more flexibility, and each has its uses. Random access iterators give immediate access to more functionality, but anything that can be done with one form of active iterator can also be done with the other (although not necessarily as efficiently). Unfortunately, random access iterators can be costly to implement for many data structures. The basic advantages and disadvantages of each iterator method are shown in Table 1.

Property	Iterator Techniques			
	Primary Operations	Passive	Active, Random Access	Active, Sequential Access
Can randomly access elements during the iteration	No*	No	Yes	No
Can access an element more than once	No	No	Yes	Yes
Can control order of visitation	No*	No	Yes	No
Can terminate at any time	Yes	No*	Yes	Yes
Preserves object structure	No	Yes	Yes	Yes
Protects object from external modification during iteration	No	Yes	Yes*	Yes*
Efficient for sequential structures	Yes	Yes	No	Yes
Does not need to know "size" of object first	Yes	Yes	No	Yes
Easy to map onto real world sequential objects (e.g, files)	Yes	Yes	No	Yes
Easy to use	No	Yes	Yes	Yes

* Indicates that a property is typical for the given iterator technique, although alternative designs are possible.

Table 1. A Comparison of Iterator Techniques

But is there a standard iterator form that can be used everywhere?[21] Ideally, a single form for iteration would promote reuse in several ways. With only one form, both the difficulty of understanding components and the difficulty of reusing components will go down. In addition, it would be easy to create generic tools that embody complex operations that are actually implemented as iterations. For example, the idea of a generic sorting operation is often conceived as a routine that iterates over a generic sequence of some kind. Figure 35, derived from [Mendal86a] and [Tracz89a], shows such an abstraction in Ada.

```
generic
    type Element is limited private;
        -- The basic operations on this type, except for Copy, are
        -- omitted for simplicity.
        with procedure Copy(source      : in     Element;
                            destination : in out Element);
        with function "<"(left, right : in Element) return boolean is <>;
        with function "="(left, right : in Element) return boolean is <>;

    type Index   is limited private;
        -- The basic operations on this type, except for Copy, are
        -- omitted for simplicity.
        with procedure Copy(source      : in     Index;
                            destination : in out Index);
        with function next_index(i : in Index) return Index;
        with function previous_index(i : in Index) return Index;

    type Sequence is limited private;
        -- The basic operations on this type are omitted for simplicity.
        with function first_index_of(s : in Sequence) return Index;
        with function last_index_of(s : in Sequence) return Index;
        with function Get_Element(from        : in Sequence;
                                  at_location : in Index) return Element;
        with procedure Put_Element(E           : in     Element;
                                   into        : in out Sequence;
                                   at_location : in     Index);

package Sort_Utilities is

    ...

end Sort_Utilities;
```

Figure 35. A Generic Sort Routine, Which Assumes the Existence of a Standardized Active Iterator for Sequences

In effect, the sort routine in Figure 35 *assumes* the existence of some standard iterator form. Notice that this routine operates on an abstract type that can be treated as a *sequence*. It sorts the elements present in this sequence, placing the elements in some user-defined order. In order to be instantiated, however, the sort routine requires the operations *first_index_of* and

21. Wheeler, David A. 1990. Institute for Defense Analyses, private communication.

last_index_of to exist for the abstract sequence, and also requires *next_index* and *previous_index* to exist for the type used as the abstract index.

Unfortunately, unless all ''container'' types—types that hold a group of elements that can be iterated over—conform to these assumptions, the reuser will have to write these operations in terms of the primitives that are provided by the container type.

In addition, a choice must be made about where to store the iterator state in order to define a standard iterator form. It is clear that sequential iterators require explicit iterator state objects if they are to ensure correct behavior for nested or concurrently active iterations. However, random access iterators often do not require explicit state objects to provide this protection, and the reduced interface complexity eases the use of such iterators. But there is a further conflict.

If an iterator allows nested or concurrent iterations, the meaning of ''destructive'' iterations becomes questionable. If one client is actively modifying the contents of a structure, it may not be meaningful for another client to currently be iterating over that structure. If both clients are actively modifying the contents without knowledge of each other, their actions could interfere to produce incorrect results. This danger is present event if concurrency protections ensure the object is always in a consistent state.

All of these concerns together create significant problems for defining a standard iterator. There currently is no clear choice for the form of a standard iterator, although sequential access iterators seem to have the most advantages of the alternatives presented here. The most popular sequential access iterators include explicit state objects and do not support destructive iterations. However, as a ''straw man'' proposal, the guidelines at the end of this section suggest standard forms for active iterators.

Another question one might ask is whether an explicit iterator should be provided, even for an abstract type that has a complete set of primary operations. As mentioned in Section 5.2.2.3, [Weide86a] recommends that the basic form of a component should only consist of its primary operators. In other words, there shouldn't be an explicit iterator because one can be built out of the available primitives. In fact, this user-built iterator would only suffer a constant factor in speed degradation over one that was provided by the component.

Although the [Weide86a] guideline is useful, it may have a negative performance impact in the long run. A lot of iterators end up being ''tight loops'' in programs. If a program spends 80% of its time in 20% of the code, and most of that 20% of the code is looping over something, then that constant factor affecting non-primitive iterators may be very important—that constant factor may turn into a constant factor in the execution time of the program. If that constant is big enough, it may be worthwhile to build the iterators into the *concept*. This reasoning is particularly true of iterative operations like *copy* and *is_equal* that may be frequently used.

Of course, the ideal solution is to provide only primary operations in the basic component, but also provide extended or enhanced versions that export the extra functionality. This way, users concerned with efficiency are not forced to pay unneeded penalties, while other users do not have to accept unneeded functionality. Because this places a greater burden on the designer to create multiple related components, though, this choice is left up to the designer.

As a result of this discussion on providing iterator capabilities for abstract data types, the following guidelines are offered:

Guideline 18:

If possible, reusable components that export abstract data types should export a *complete* set of primary operations, so that the reuser can construct arbitrary iterations using these primary operations and the language mechanisms available for that purpose.

Guideline 19:

If a component designer chooses to export an explicit iterator, active, sequential access iterators are preferred over passive iterators. Such an iterator should protect objects of the abstract data type from structural modification while it is being iterated over. In addition, such an iterator should support destructive capabilities. and should also use explicit iterator state objects to ensure correct behavior under nested or concurrent iterations. The following profile is recommended:

```
type iterator_state is limited private;

procedure start_iteration(object       : in out ADT;
                          state        : in out iterator_state);
function being_iterated_over(object : in ADT) return boolean;
procedure access_current_entry(object       : in out ADT;
                               state        : in out iterator_state;
                               element      : in out element_type);
    -- "Swaps" the current value of ELEMENT with the value of the
    -- subpart of OBJECT located at the current position in the iteration STATE.
procedure advance(object     : in out ADT;
                  state      : in out iterator_state);
function iteration_is_complete(object       : in out ADT;
                               state        : in out iterator_state) return boolean;
```

```
procedure end_iteration(object      : in out ADT;
                        state       : in out iterator_state);

iterator_error : exception;
operation_not_allowed : exception;
```

Figure 36. Suggested Standard Profile for an Active, Sequential Iterator

Guideline 20:

If a component designer chooses to export an active, random access iterator, it should be provided *in addition to*, rather than in lieu of, an active, sequential access iterator. This will facilitate the development of tools that encapsulate iterative processes and that expect a common set of operations for performing such iterations. When a random access iterator is included, it should use explicit iterator state objects, support destructive iterations, and ensure correct behavior under both nested and concurrent iterations. The following profile is recommended:

```
type iterator_state is limited private;

procedure start_iteration(object      : in out ADT;
                          state       : in out iterator_state);
function being_iterated_over(object : in ADT) return boolean;
function size_of(object : in ADT) return integer;
procedure access_nth_entry(object        : in out ADT;
                           state         : in out iterator_state;
                           position      : in     integer;
                           element       : in out element_type);
    -- "Swaps" the current value of ELEMENT with the value of the
    -- subpart of OBJECT located at POSITION. POSITION can be in the
    -- range 1 .. SIZE_OF(M).
procedure end_iteration(object : in out ADT;
                        state : in out iterator_state);

iterator_error : exception;
operation_not_allowed : exception;
```

Figure 37. Suggested Standard Profile for an Active, Random Access Iterator

Note that guidelines 19 and 20 only provide suggestions for standard iterator profiles. Also notice that the "access" operations in both profiles do not imply that data is *copied* out of the structure during an iteration; instead data is swapped out. This is in contrast to the *value_of* routine in Figure 33, which forces data to be duplicated. Also, the names of the operations in these two suggested profiles are not important. It is the functionality of the operations, number of arguments, and placement of arguments that is important for composability.

6.4 SAVE/RESTORE BEHAVIOR

One of the more obscure, but also often difficult, problems occasionally encountered by reusable component designers has to do with *save* and *restore* operations. It is occasionally useful for a component to export these operations for a given exported abstract type.

Some abstract types hold information that the reuser might naturally need to preserve from one invocation of a program to another. For example, a programmer creating a rapid prototype of a new compiler might choose the GPD structure introduced in Section 3 to contain a representation of an abstract syntax tree. If the compiler also supported a significant library management system outside of the compiler (like Ada compilers do), the programmer might wish to store this abstract syntax tree as an intermediate representation of compiled source code in his library. DIANA, an intermediate representation for compiled Ada programs, is used in much the same way.

A component designer, anticipating these needs, might wish to provide the capability of moving abstract objects to secondary storage in his component. Choosing a strategy for providing this functionality raises some of the same basic alternative solutions as choosing a memory management (Section 6.1) or concurrency protection (Section 6.2) strategy. In particular, the designer can

a. exclude this capability (as in **unmanaged** memory management or **sequential** concurrency control),

b. require the user to explicitly participate in providing the capability (as in **explicitly managed** memory management, or **shared** or **guarded** concurrency control),

c. or implement the capability in a completely transparent fashion (as in **transparently managed** memory management, or **concurrent** or **multiple** concurrency control).

This results in 3 main possibilities for the storage model supported by a component for a particular abstract type:

a. **Transient** abstract data types have no predefined facility for moving them to or from secondary storage, although it might be possible to build these mechanisms from the primitives provided for the type.

b. **Storable** types have explicitly exported operations that allow them to be copied to or from secondary storage. These operations usually take the form of a *save* operation and a *restore* operation.

c. Persistent types can be moved to or from secondary storage by the component, although this movement is transparent to the reuser. Thus, objects of **persistent** types appear to the reuser to continue to exist between program executions. Note that **persistent** types can be further divided into:

   (1) **Explicitly persistent** types allow both **transient** and **persistent** objects of that type to exist. The user must specify in some manner which objects of the type are **persistent** and which are **transient**. Furthermore, when a **persistent** object is no longer needed, the user must explicitly state so (via some kind of *free* operation).

   (2) **Implicitly persistent** types are implemented so that all objects of the type are considered to be **persistent**. The persistence mechanism used in the component's implementation is responsible for determining when *persistent* objects can no longer be accessed by programs, and then reclaiming the resources used by such objects [Wileden88a].

The issue of constructing effective, reusable components that are **persistent** is currently open, and will not be discussed here. However, interested readers should refer to [Wileden88a] for an introduction to reusable components that provide this capability.

In this section, it is the **storable** components that are of interest. Notice that in fact, the *save* and *restore* operations provided by such components are both just special forms of iterators. As such, one may argue that they need not even be provided in a "complete" component, since they can be constructed from primary operators. But because they may be extremely cumbersome for the reuser to implement in terms of primitive operators (or even in terms of explicit iterator constructs), they can be prime candidates for inclusion in an "extended" version of a component. Unfortunately, they may cause unique problems that do not occur in the majority of iterators.

To see how these problems manifest themselves, consider adding *store* and *retrieve* operations to the GPD concept introduced in Section 3 and modified throughout Section 5. The first attempt at such an addition might appear as shown previously in Figures 23 (p. 61) and 25 (p. 62) of Section 5.2.3.

Unfortunately, this interface will not work, in general, for structurally shared objects. Consider a GPD structure, as extended in Section 5.2.3, that may contain references to data the may be structurally shared. For example, there may be a user-defined GPD node type that contains references to separate linked list structures. Figure 38 illustrates an example structure of this type. Notice that *node_1* and *node_3* both point into the same shared structure.

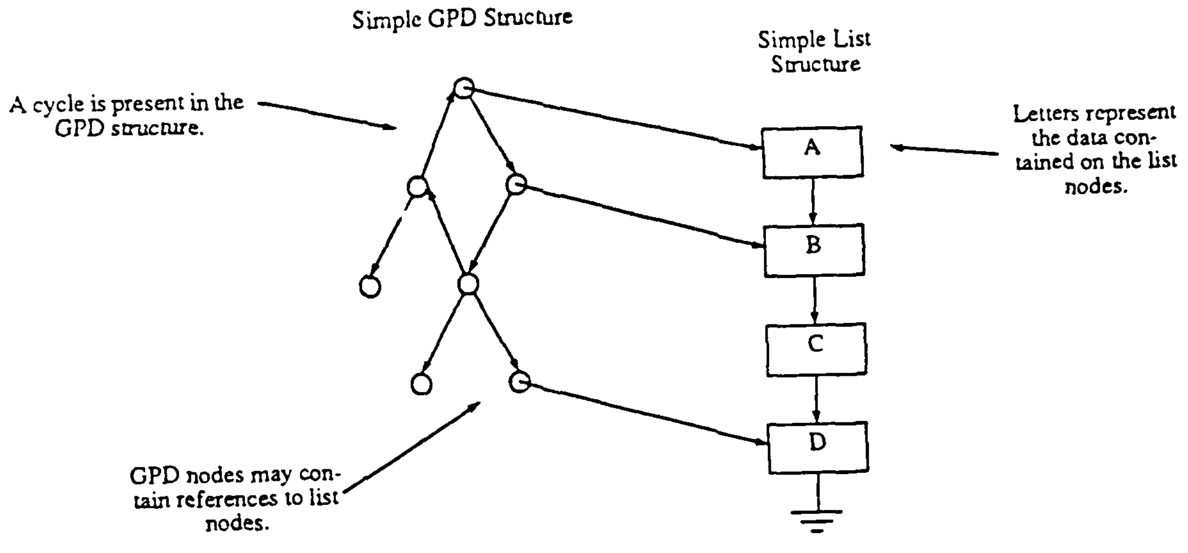

Simple GPD Structure
Simple List Structure
A cycle is present in the GPD structure.
Letters represent the data contained on the list nodes.
A
B
C
D
GPD nodes may contain references to list nodes.

Figure 38. A GPD Structure That Refers to Structurally Shared Data

A standard implementation of the *save* operation for GPD structures would probably look similar to the Ada-like pseudocode presented in Figure 39. Unfortunately, when combined with a typical *read* operation, it would produce incorrect results. A *save* operation on the structure shown in Figure 38, followed by a *read* operation to restore that data would result in Figure 40. Although the structural sharing within the GPD itself was faithfully reproduced, information about structural sharing in either the *Common_Node_Contents* or any other user-defined data type stored in the GPD was lost. The reproduction in Figure 40 does not include the same structural sharing in the lists contained within the GPD structure.

This is a result of the fact that the same routine is being used both to save a single, isolated object, and to save a collection of objects related through structural sharing. In order to resolve and reproduce the structural relationships between a group of objects with shared subparts, a different approach is necessary. The information used to recover structural sharing within a single object may also be required during the storage of subsequent objects in order to recover inter-object sharing.

```
procedure save(file      : in text_io.file_type;
               gpd       : in gpd_type);
begin
    -- pass one
    mark_nodes(root);          -- Symbolically label all the nodes that can be
                               -- reached from the root node. This includes setting
                               -- up the information required to store structurally
                               -- shared objects. If more than one GPD will be stored,
                               -- it should also be marked here.
    -- pass two
    for node in (gpd_structure) loop
        -- Do this for every node that can be reached from the root,
        -- probably via a depth-first recursive procedure rather than
        -- an actual for loop. If more than one GPD structure
        -- is being stored, this loop should iterate over all of them
        if not_visited(node) then -- detect cycles
            -- First, write out the node's structure, including all
            -- information common to all classes of nodes
            write_basic_node_state(file, node);
            -- Now write out the Common_Node_Contents of the node via
            -- the write procedure passed in by the user during instantiation
            write(node.contents);
            case node_class_of(node) is
                when user_defined =>
                    -- Invoke the table-driven execution mechanism to
                    -- call the write operation provided by the user that
                    -- corresponds to the type of the data in this node.
                    -- This operation is represented by the following
                    -- call:
                    call(procedure_name      => write,
                         arguments           => (file, node.user_contents),
                         for_type            => node.user_type);
                when others =>
                    -- Write out information specific to this
                    -- node class (ommitted for simplicity).
            end case;
        end if;
    end loop;
    -- pass three
    unmark_nodes(root);          -- Clean up any information left around to
                                 -- maintain the map from nodes to their symbolic names.
                                 -- This step should be repeated for each GPD
                                 -- structure that is being stored, just as during
                                 -- the first pass.
    -- Any other cleanup steps go here.
end save;
```

Figure 39. High-Level Pseudocode for the Save Operation

To implement this form of saving, it is necessary to iterate over all the objects to be saved, keeping any temporary information about inter-object structural sharing available during the whole iteration. Perhaps the most readily apparent approach is to create a three pass algorithm, as shown in Figure 39. Further thought can produce an equivalent two pass version, where

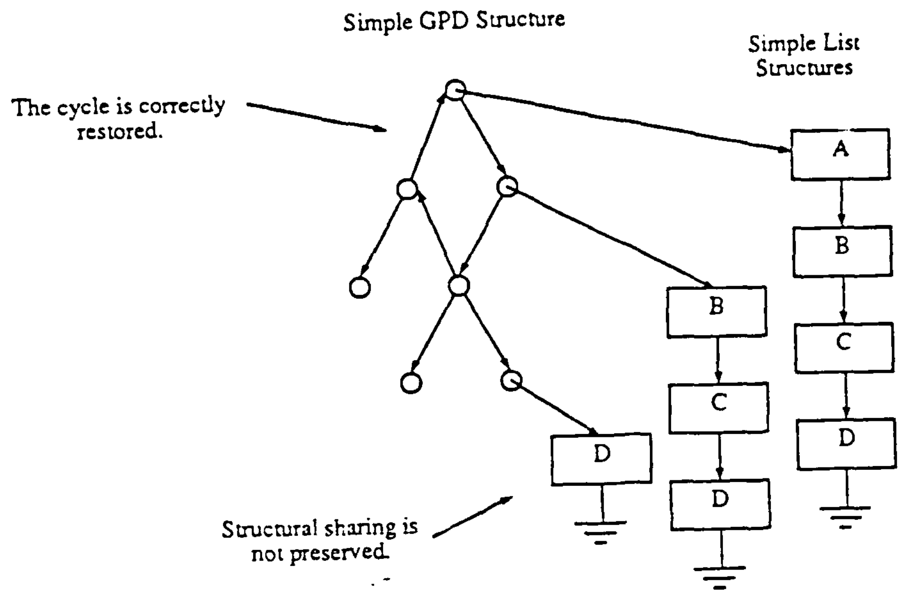

Simple GPD Structure
Simple List Structures
The cycle is correctly restored.
A
B
C
D
B
C
D
D
Structural sharing is not preserved.

Figure 40. Erroneously Reproduced Data Structure

the "marking" is rolled into the actions performed the first time a node is visited. Also, the final pass for "unmarking" can be optimized so that a visit to each node need not take place.

Unfortunately, although three passes in Figure 39 are actually made to store the GPD structure, actions are only performed on the user-defined subparts during the second phase. Another important concern is whether or not the abstraction supplied as a user-defined subpart, a list in this example, actually supports storage of groups of structurally interconnected objects in addition to single objects. While assumptions about the generic types used to tailor a reusable component often make that component's implementation easier, they can also implicitly reduce reusability. In this case, the assumption that those generic parameters can be stored in a single pass is implicit in the description of those parameters, and in the implementation of the GPD *save* and *restore* routines. Furthermore, this assumption restricts the abstractions the reuser can supply for these generic type parameters in order to instantiate his GPD component, and thus reduces the reusability of the unit.

Solving this problem once it is recognized appears to be relatively easy. It is simple to alleviate the assumption that objects are stored in isolation, which is present in the generic parameters. One possible solution is shown in Figure 41, which depicts the changes to the generic parameters of the *GPD_pkg* necessary to support storage of structurally shared

Common_Node_Contents. This solution implements a two pass interface, with the final "unmarking" pass hidden inside the *finish_writing_group* operation.

```
...
generic
    type Common_Node_Contents is limited private;
    -- basic operations omitted for simplicity

    with procedure prepare_to_read_group(
        file : in text_io.file_type);
        -- This routine is called before a group of objects of type
        -- Common_Node_Contents will be read from the specified FILE.
        -- This opportunity can be used to load any information that was
        -- previously stored about the group as a whole.
    with procedure read_one_of_a_group(
        file     : in     text_io.file_type;
        data     : in out Common_Node_Contents);
        -- The DATA parameter of READ is mode in out because READ
        -- FINALIZEs the incoming value of DATA before placing the
        -- result of the READ operation in it.
    with procedure finish_reading_of_group;
        -- This routine is called once the reading of a group of objects
        -- of type Common_Node_Contents has been completed. This allows
        -- any internal state used by the component defining the
        -- type Common_Node_Contents during the "reconstruction" operations
        -- to be cleaned up.

    with procedure prepare_to_mark_group;
        -- This routine is called before a group of objects of type
        -- Common_Node_Contents will be written to a file. It allows any
        -- internal state within the Common_Node_Contents component that will
        -- be used for representing structural sharing to be initialized.
    with procedure mark_for_writing(
        data : in Common_Node_Contents);
        -- This is called during the first pass of the saving operation
        -- to indicate that this data element will be written to a file.
    with procedure marking_completed_prepare_to_write_group(
        file : in text_io.file_type);
        -- This routine is called after all elements of the group have
        -- been marked, but before the group has been written to the file.
        -- It allows the component defining Common_Node_Contents to write
        -- out any internal state information about the group as a whole
        -- that will be useful in reconstructing all the pieces.
    with procedure write_marked_element(
        file     : in text_io.file_type
        data     : in Common_Node_Contents);
        -- In effect, this routine sends a "copy" of DATA to FILE.
    with procedure finish_writing_group;
        -- This routine is called once the writing of a group of objects
        -- of type Common_Node_Contents has been completed. This allows
        -- any internal state used by the component defining the
        -- type Common_Node_Contents during the storage operations
        -- to be cleaned up.

package GPD_pkg is
```

```
...
```

Figure 41. Modifications to the Generic Parameters of *GPD_pkg* to Support Two-Pass Saving

There are problems with standardizing on this approach, however. First, the choice of how many passes to support to important. Second, the specification of the operations should allow concurrency-hardened versions of the operations to be created. In other words, it should be possible for two threads of control to be simultaneously storing two different objects to independent locations, without interfering with each other.

Currently, there is little work available on the issues of general-purpose, composable skeletons for ''save'' and ''restore'' operations. This section proposes a potential candidate, although it is unproven. To address the first concern, the use of a single explicit pass is recommended. As noted previously, the first ''marking'' pass can be rolled into the second pass. If a ''map'' of unique object identifiers to symbolic identifiers is used as the marking scheme, the final pass can be hidden in the ''finish'' operation[22]. While sets of ''save'' and ''restore'' operations that conform to this specification may be more difficult to write, they are feasible. Instead, it is the reduced complexity of the interface that guides this recommendation.

To address the concern about concurrently executed ''saves,'' the solution presented in Section 6.3.4 will also be recommended here—the set of operations implementing ''save'' and ''restore'' behavior for groups of objects should incorporate explicit iterator state objects. This will ensure that the ''marking'' information appropriate for one invocation of an operation is not confused with that of another concurrent invocation.

Figures 42 and 43 illustrate the changes necessary to the *GPD_pkg* to conform with these recommendations. Note that similar modifications would have to be made to the *user_defined_node_pkg* (Figure 25, p. 62) as well. Similar modifications would also be necessary in the hypothetical list concept discussed as part of this example.

Guideline 21:

If possible, components that support save and restore behavior for the abstractions they provide should follow the example in Figure 43 for operations exported by the component. Such components should also follow the example in Figure 42 for operations on generic formal type parameters.

22. Wheeler, David A. 1990. Institute for Defense Analyses, private communication.

```
...
generic
        type Common_Node_Contents is limited private;
        -- basic operations omitted for simplicity

        -- For saving and restoring a single object of type Common_Node_Contents:
        procedure save(file      : in text_io.file_type;
                       data      : in Common_Node_Contents);
        procedure restore(file     : in          text_io.file_type;
                          data     : in out Common_Node_Contents);

        -- For saving and restoring a group of Common_Node_Contents objects that might be
        -- interconnected (in order to recover the interconnections):

        type CNC_Save_State is limited private;
        -- basic operations omitted for simplicity
        type CNC_Restore_State is limited private;
        -- basic operations omitted for simplicity

        with procedure prepare_to_read_group(
                file                : in      text_io.file_type;
                read_state          : in out CNC_Restore_State);
                -- This routine is called before a group of objects of type
                -- Common_Node_Contents will be read from the specified FILE.
                -- This opportunity can be used to initialize the READ_STATE,
                -- and then load any information that was previously stored about
                -- the group as a whole into the into that state variable.
        with procedure read_one_of_a_group(
                file                : in      text_io.file_type;
                read_state          : in out CNC_Restore_State;
                data                : in out Common_Node_Contents);
                -- The DATA parameter of READ is mode in out because READ
                -- FINALIZEs the incoming value of DATA before placing the
                -- result of the READ operation in it.
        with procedure finish_reading_of_group(
                read_state : in out CNC_Restore_State);
                -- This routine is called once the reading of a group of objects
                -- of type Common_Node_Contents has been completed. This allows
                -- the READE_STATE to be cleaned up and finalized.

        with procedure prepare_to_write_group(
                write_state : in out CNC_Save_State);
                -- This routine is called before a group of objects of type
                -- Common_Node_Contents will be written to a file. It allows the
                -- WRITE_STATE, used for representing structural sharing
                -- information, to be initialized.
        with procedure write_marked_element(
                file                : in      text_io.file_type
                write_state         : in out CNC_Save_State;
                data                : in      Common_Node_Contents);
                -- In effect, this routine sends a "copy" of DATA to FILE.
        with procedure finish_writing_group(
                write_state : in out CNC_Save_State);
                -- This routine is called once the writing of a group of objects
                -- of type Common_Node_Contents has been completed. This allows
                -- the WRITE_STATE to be cleaned up and finalized.
```

```
package GPD_pkg is
. . .
```

Figure 42. Recommended Changes to the Generic Parameters of the *GPD_pkg* to Support Save and Restore Behavior

```
. . .
package GPD_pkg is
    . . .

    -- For saving and restoring a single GPD structure:
                            gpd  : in gpd_type);
    procedure restore(file      : in      text_io.file_type;
                      gpd       : in out gpd_type);

    -- For saving and restoring a group of GPD structures that might be
    -- interconnected (in order to recover the interconnections):

    -- Saving:
    type GPD_save_state is limited private;
    -- basic operations omitted for simplicity
    type GPD_restore_state is limited private;
    -- basic operations omitted for simplicity
    procedure prepare_to_write_group(
        save_state : in out GPD_save_state);
    procedure write_marked_element(
        file            : in      text_io.file_type;
        save_state      : in out GPD_save_state;
        data            : in      Common_Node_Contents);
    procedure finish_writing_group(
        save_state : in out GPD_save_state);

    -- Restoring:
    procedure prepare_to_read_group(
        file            : in      text_io.file_type;
        read_state      : in out GPD_save_state);
    procedure read_one_of_a_group(
        file            : in      text_io.file_type;
        read_state      : in out GPD_save_state;
        data            : in out gpd_type);
    procedure finish_reading_of_group(
        read_state : in out GPD_save_state);

. . .
```

Figure 43. Recommended Changes to the Operations Exported by the *GPD_pkg* to Support Save and Restore Behavior

7. CONCLUSIONS

Section 2 of this paper argues that reduced maintenance cost should be the dominant goal of software reuse. While others have cited the potential for software development savings, the potential for maintenance savings are far more dramatic. Sections 2.6 and 5.3 raise a significant problem that has not yet been solved in any language. It is often said that the biggest reuse benefits are observed when the largest pieces are reused, yet the parameterization management problem these sections discuss may indicate that these benefits from large granularity reuse will not materialize without the development of new techniques. This paper identifies and describes the problem, without providing a solution. More research on this issue is needed.

Sections 4 through 6 discuss the use of Ada's features to represent reusable components, providing guidelines to provoke the component author into considering how use of these features will affect reusability. These guidelines are strongly based on the 3C model presented in Section 2. Although this model is still developing, it is very useful as a tool for thinking about the process of reusing software, and the process of designing reusable components. It can help to shed light on both the difficulties of these processes and ways to solve those difficulties. This paper captures some of the current knowledge that has been revealed by this model about the construction of software components in Ada, in the belief that this information will aid other software engineers in constructing software that is more reusable.

In addition to these general issues, this paper also presents a collection of knowledge about building reusable software components in Ada. This knowledge, gathered through experience, is valuable to the Ada component designer attempting to design for reuse. With or without the 3C model, it provides a look at some of the problems that can arise when design decisions are made implicitly, without considering their reuse ramifications.

Thus, the body of the paper serves to educate the prospective component designer about these issues. In addition, the guidelines presented in the paper and summarized in Appendix B can be used by designers as a check list for design decisions affecting reuse. Together with the preliminary component labeling scheme presented in Appendix A, the guidelines will encourage component designers to continually ask questions about how the decisions they make about a particular component will affect its reusability. It is this spirit of concern that will lead to more reusable software.

APPENDIX A

ATTRIBUTES FOR LABELING THE INTERFACE CHARACTERISTICS OF REUSABLE ADA COMPONENTS

This Appendix contains a list of "attributes" which can be used to characterize reusable components, as well as different implementations of a single reusable component. This list is primarily an extension of the "forms" used by Booch to differentiate the behavior of different implementations of a single *concept* [Booch87a, pp. 40-43]. These attributes are designed to capture the information necessary to determine the applicability and ease-of-use of a reusable component once it has been selected, not to define a taxonomy of components for the purpose of easy component retrieval.

This list is also comprehensive enough to cover components which do not follow the guidelines presented in this paper. In fact, if a component is given attribute values according to this list, decisions about the design of its interface which diverge from the guidelines presented in this paper will be readily apparent.

This list is primarily suited toward categorizing components which define abstract data types (ADTs). Each implementation of an ADT-exporting component will have some value for each attribute. The attributes are grouped into the following major and minor categories based on the function of the information they provide:

a. Implementation Characteristics:

   (1) Memory Management

   (2) Concurrency Protection

b. Characteristics of Generic Parameters to the Component:

   (1) Class of the Generic Parameter

   (2) Initialization/Finalization for the Generic Parameter

   (3) Data Movement for the Generic Parameter

   (4) Implied Semantics of Exported Operations

   (5) Imported Iterator

(6) Save/Restore Operations

c. Characteristics of Exported Types and Operations:
Class of the Exported ADT

(1) Boundedness of the Exported ADT

(2) Initialization/Finalization for the Exported ADT

(3) Data Movement for the Exported ADT

(4) Exported Iterator

(5) Save/Restore Operations

Each category, along with its associated attributes, will be presented in turn, including the set of possible values for each attribute.

1. IMPLEMENTATION CHARACTERISTICS

a. Memory Management

(1) Is memory space reclamation handled by the component, or left up to the language run-time system or operating system?

  i) **Unmanaged** components provide no mechanism for space reclamation, usually relying on the underlying run-time system to provide automatic garbage collection.

  ii) **Managed** components do provide a mechanism for space reclamation.

(2) Does the user actively participate in the memory management process? (This question only applies to **managed** components.)

  i) **Explicitly** managed components require the user to call some form of "free" or "deallocate" routine in order to explicitly reclaim space.

  ii) **Transparently** managed components do not require any explicit action on the user's part to "free" unneeded memory space. Instead, they usually implement some local form of garbage collection within the component. Such components often rely on the user to consistently call the appropriate *finalize* operations on variables when they are leaving scope.

(3) Is the memory management portion of the component protected against use by concurrently executing threads of control? (Again, this question only applies to **managed** components.)

  i) **Controlled** components are built so that exported *free* or *finalize* routines work correctly even if simultaneously executed by multiple threads of control.

  ii) **Uncontrolled** components require the corresponding *free* and *finalize* routines to only be executed by one thread of control at a time.

(4) Are there limits on the total amount of memory that can be used by the component? (This question applies to both **managed** and **unmanaged** components.)

i) **Limited** components have an upper bound on how much memory can be consumed by all of the active objects of the exported abstract data type at one time. For example, a linked list abstraction may allow the reuser to set an upper limit on the total amount of memory that may be used by all of the lists that are currently allocated.

ii) **Unlimited** components do not allow the reuser to place a bound on their memory consumption.

(5) Concurrency Protection

i) **Sequential** components provide no concurrency protection whatsoever.

ii) **Shared** components provide concurrency protection for the internal state, both hidden and visible portions, of the package (or instantiation) exporting the abstraction (for example, internal hash tables, caches, etc.). **Shared** components do not, however, provide any built in protection for objects of the exported abstract type(s). Multiple threads of control can "share" the facility, although they must provide some other protection mechanism of their own to share objects from that facility.

iii) **Guarded** components ensure that each object of an exported abstract type has some form of abstract "lock," but multiple threads accessing a shared object must obey the associated "locking" conventions. In other words, the necessary framework for mutual exclusion at the object level is provided, but it is up to the client tasks to assure that mutual exclusion is maintained by obeying the conventions of this framework. Note that **guarded** components also provide the component state protections of **shared** components.

iv) **Concurrent** components guarantee that each object of an exported abstract type may only be accessed by one thread at a time. The component assumes responsibility for ensuring mutual exclusion, regardless of the actions of the client tasks. This makes the framework of mechanisms for ensuring mutual exclusion transparent to the user (client). Note that **concurrent** components also provide the component state protections of

shared components.

v) **Multiple** components have the same responsibility as **concurrent** components, ensuring that objects of exported types and the component itself maintain consistent states in the face of arbitrary, concurrent accesses. The difference is that **multiple** components optimize accesses to a single object (or to the shared internal state of a component) for maximal concurrency, for example, allowing multiple, simultaneous read operations but only allowing exclusive write operations.

2. CHARACTERISTICS OF GENERIC PARAMETERS TO THE COMPONENT

These attributes will help a potential reuser to decide whether or not he can instantiate the component correctly. For example, these attributes will help a potential reuser to determine whether the operations exported by his Stack component will allow him to use it as a parameter to your List component. For each abstract type in the *context* of the component, the following list of attributes should be given.

a. Class of the Generic Parameter

(1) **Standard** parameters are generic formal type parameters that are specified as generic type definitions [DoD83a, Section 12.12], rather than as a **private** or **limited private** type declaration. Only matching Ada types can be provided for such a parameter, so no opaque types can be provided for this parameter by the reuser.

(2) as being **private** in the generic formal parameter list of the Ada specification for the component. An Ada type which is **limited private** cannot be provided as a value for this parameter when the component is instantiated.

(3) are declared as being **limited private** in the generic formal parameter list of the Ada specification for the component. Any Ada type can be provided as a value for this parameter when the component is instantiated.

b. Initialization/Finalization for the Generic Parameter

(1) **None**—this label indicates that no mechanisms other than the **private/limited private** nature of the generic formal declaration is used to support encapsulation of this parameter to the reusable component.

(2) **Initialization only** indicates that a separate initialization routine is imported along with the corresponding type parameter. No mechanism for finalizing data items of this type is provided, however.

(3) **Finalization only** indicates that a separate finalization routine is imported along with the corresponding type parameter. No mechanism for initializing data items of this type is provided, however.

(4) **Initialization/Finalization** indicates that both initialization and finalization procedural parameters are imported in addition to the type definition to support complete encapsulation of the corresponding type.

c. Data Movement for the Generic Parameter

(1) **Standard** indicates that the component relies on the standard Ada mechanisms for moving data of this user-defined type. If the corresponding type parameter is **limited private**, no data movement for that types is performed inside the component. If the corresponding type parameter is **private**, then the predefined Ada assignment operator is used to copy objects of the corresponding type inside the component.

(2) **Copy** indicates that the component relies on a user-defined *copy* procedure, also passed into the component as a formal parameter along with the corresponding type, to move data.

(3) **Swap** indicates that the component relies on a user-defined *swap* procedure, also passed into the component as a formal parameter along with the corresponding type, to move data.

d. Implied Semantics of Exported Operations

(1) **Copy** indicates that the component either exports a function that returns a value of the corresponding type, or performs operations that imply the duplication of objects of the corresponding type. If the component exports operations with **copy** semantics, and the corresponding type parameter is **standard**, the reuser should take this as a warning that the component may behave differently, depending on whether or not the type he provides for the corresponding parameter supports structural sharing.

(2) **Swap** indicates that the component does not export any operations that imply or require the duplication of objects of the corresponding parameter type.

e. Imported Iterator

(Note that passive iterators are generics, and therefore cannot be passed into an Ada package as a generic parameter. Such constructs are not represented in this categorization.)

(1) **None** indicates that the component does not need to iterate over the subparts of any object of the corresponding parameter type.

(2) **Primary** indicates that the component imports enough primary operations along with the corresponding type parameter so that it can construct iterators.

(3) **Sequential access** indicates that all of the operations that are part of an active, sequential access iterator are imported as generic parameters along with the corresponding type parameter.

(4) **Random access** indicates that all of the operations that are part of an active, random access iterator are imported as generic parameters along with the corresponding type parameter.

f. Save/Restore Operations

(1) **None** indicates that no save/restore operations are imported for the corresponding type parameter.

(2) **One-pass** indicates that appropriate formal subprogram parameters are imported along with the corresponding type parameter to provide for a one-pass set of save/restore operations for the corresponding type. The reuser should be aware that he may not be able to supply a type that requires two pass save/restore operations as the value of the corresponding type parameter when instantiating this component.

(3) **Two-pass** indicates that appropriate formal subprogram parameters are imported along with the corresponding type parameter to provide for a two-pass set of save/restore operations for the corresponding type.

3. CHARACTERISTICS OF EXPORTED TYPES AND OPERATIONS

These attributes will help a potential reuser to decide whether or not he can instantiate the component correctly. For example, these attributes will help a potential reuser to determine whether the operations exported by his Stack component will allow him to use it as a parameter to your List component. For each abstract type in the *context* of the component, the following list of attributes should be given.

a. Class of the Exported ADT

(1) **None** indicates that the corresponding exported type is not declared as a **private** Ada type in the package specification for the component. This exported type is completely open. This classification for an exported type is a warning sign that the type is not abstracted or encapsulated.

(2) **Private** indicates that the corresponding exported type is declared as a **private** Ada type. The comparison and assignment operators for this type provide correct semantics. This classification for an exported type is a warning sign that the type is not sufficiently encapsulated.

(3) **Aliased private** indicates that the corresponding exported type is declared as a **private** Ada type. The comparison or the assignment operator for this type does *not* provide correct semantics. This classification for an exported type is a warning sign that the type is not sufficiently abstracted or encapsulated.

(4) **Limited private** indicates that the corresponding exported type is declared as a **limited private** Ada type, and is strongly encapsulated.

b. Boundedness of the Exported ADT

(There are many different possibilities for the size behavior of ADTs. This portion of the labeling scheme is merely a placeholder for future work in the effective labeling of an attribute such as "boundedness.")

(1) **Bounded** indicates that the size of an object of the corresponding exported type is static.

(2) **Unbounded** indicates that the size of an object of the corresponding exported type is dynamic.

c. Initialization/Finalization for the Exported ADT

(1) **None None**—this label indicates that no mechanisms for initialization or finalization are explicitly exported from the component. Exported types with this label should be designed so that *neither* initialization nor finalization for objects of that type is necessary.

(2) **Initialization only** indicates that a separate initialization routine is exported along with the corresponding type. No mechanism for finalizing data items of this type is provided, however. All users of this type must call this initialization operation on newly created objects of the corresponding type to ensure correct behavior.

(3) **Finalization only** indicates that a separate finalization routine is imported along with the corresponding type parameter. No mechanism for initializing data items of this type is provided, however. All users of this type must call this finalization operation on objects of the corresponding type as they leave scope to ensure correct behavior.

(4) **Initialization/Finalization** indicates that both *initialization* and *finalization* operations for the corresponding type are exported by the component. All users of the component must faithfully call these operations as objects of the corresponding type are created or leave scope to ensure proper behavior of the component.

Data d. ADT

(1) **Standard** types have no data movement mechanism explicitly provided by the component. If the type is **private**, the reuser can use predefined assignment for moving objects of the type. If the type is **limited private**, the reuser cannot move objects of the type.

(2) **Copy** indicates that an explicit copy operation (assignment) is provided for the type by the component.

(3) **Swap** indicates that a swap operation is provided for type by the component for moving objects of the type.

e. Exported Iterator

(If more than one form of iterator is provided for the type, label with all applicable labels.)

(1) **Noniterator** types have no iteration capability whatsoever.

(2) **Primary** types have no explicit iterator construct, but an iteration can be constructed from the primary operations available on the type.

(3) **Passive** types have an explicitly declared passive iterator that is exported by the component.

(4) **Sequential access** types have an explicitly declared, active, sequential access iterator that is exported by the component.

(5) **Random access** types have an explicitly declared, active, random access iterator that is exported by the component.

f. Save/Restore Operations

(1) **None** indicates that no operations are exported for saving or restoring objects of the corresponding type to or from secondary storage.

(2) **One-pass** indicates that objects of the corresponding type can be saved in a one-pass operation, and that the type does not allow explicit structural sharing. The component exports appropriate subprograms for saving or restoring single objects of the corresponding type.

(3) **Two-Pass** indicates that objects of the corresponding type require two passes to save, and that groups of such objects may have structural sharing relationships. As a result, the component exports two sets of save/restore operations. In addition to single-object save/restore operations, the component also exports the appropriate subprograms to provide a two-pass, multiple object set of save/restore operations. This second set of operations will allow groups of interconnected objects to be saved and restored while maintaining those interconnections.

4. USING THE ATTRIBUTE VALUES

The ramifications of each attribute are often interconnected, so that attributes must be viewed in combination to determine the suitability of an implementation for any given application. Some common concerns and the attributes which affect them are:

a. Concurrent usage:

   (1) Concurrency Protection

   (2) Memory Management

b. Efficiency for large data types:

   (1) Class of Generic Parameters

   (2) Data Movement for Generic Parameters

   (3) Implied Semantic., of Exported Operations

c. Encapsulation of parameters (i.e., what are the limits on the abstract types that are provided as parameters to this component during instantiation?):

   (1) Class of Generic Parameters

   (2) Initialization/Finalization for Generic Parameters

   (3) D^'a Movement for Generic Parameters

To see how these attributes can be applied to a real component, consider the GPD *concept* discussed throughout the paper. The final form of this unit is presented in Appendix C. This component has one primary type parameter, *Common_Node_Contents*, and one exported abstract type, *gpd_type Applying* the *labels* discussed the following information:

a. Implementation Characteristics:

   (1) Memory Management: **Explicitly Managed, controlled, unlimited.**

   (2) Concurrency Protection: **Shared.**

b. Characteristics of Generic Parameter—*Common_Node_Contents*:

(1) Class: **Limited private.**

(2) Initialization/Finalization: **Initialization/Finalization.**

(3) Data Movement: **Swap.**

(4) Implied Semantics of Exported Operations: **Swap.**

(5) Imported Iterator: **None.**

(6) Save/Restore Operations: **Two-pass.**

c. Characteristics of Exported Types and Operations—*gpd_type*:
Class: **Limited private.**

(1) Boundedness: **Unbounded.**

(2) Initialization/Finalization: **Initialization/Finalization.**

(3) Data Movement: **Swap.**

(4) Exported Iterator: **None.**

(5) Save/Restore Operations: **Two-pass.**

Also, it might be useful to consider the labeling that would result from following all of the guidelines presented in this paper. This would generate the following:

a. Implementation Characteristics:

(1) Memory Management:

Components that do not export dynamic types should be **unmanaged.** Components that export dynamic types should ideally be **explicitly managed** and **controlled.** Components be either **limited** or **unlimited,** at the designer's discretion.

(2) Concurrency Protection:

Components that are not designed for concurrent applications may be either **sequential** or **shared.** Components that are designed for such applications may be either **concurrent** or **multiple. Guarded** components should be avoided, if possible.

b. Characteristics of Generic Parameters to the Component:

(1) Class of the Generic Parameter:

Limited private is the best choice for type in the *context* of a component, and other types should be avoided.

(2) Initialization/Finalization for the Generic Parameter:

Initialization/Finalization is needed for general support of completely encapsulated types.

(3) Data Movement for the Generic Parameter:

Swap is preferred, since it does not suffer from the efficiency problems that **copy** can impose on large data types.

(4) Implied Semantics of Exported Operations:

Swap is preferred here as well, again because of efficiency concerns.

(5) Imported Iterator:

Components that do not require iteration capabilities can be labeled as **None**. Components that do require these capabilities should be **sequential access**, *if* possible.

(6) Save/Restore Operations:

Components that do not require save/restore capabilities can be labeled as **None**. This should include all components that only supply basic functionality. However, extended components that support save/restore behavior and thus require these capabilities should be **two-pass** for maximum generality.

c. Characteristics of Exported Types and Operations:
Class of the Exported ADT:

Limited private is the best choice for exported types, and other forms should be avoided.

(1) Boundedness of the Exported ADT:

Any label is acceptable.

(2) Initialization/Finalization for the Exported ADT:

Again, **Initialization/Finalization** is needed for general support of completely encapsulated types.

(3) Data Movement for the Exported ADT:

Again, **swap** is preferred, since it does not suffer from the efficiency problems that **copy** can impose on large data types.

(4) Exported Iterator:

If possible, **primary** is preferable. If the designer desires an explicit iterator, **sequential access** is the best choice.

(5) Save/Restore Operations:

None, on basic components if an appropriate iterator can be built to perform the task from the primary operations. Extended components that support more capabilities over the primary operations may include *save* and *restore* operations, however. Such components should meet the criteria for **two-pass labeling**.

APPENDIX B

SUMMARY OF GUIDELINES

This Appendix repeats the individual guidelines introduced in the main body of this paper. These guidelines are focused on "average case" reusable components. Components designed specifically for a real-time, embedded, or otherwise restricted application may require different choices. To assist review, the page number where each guideline originally appeared is presented along with that guideline in the summary. The reader working on application-specific components should refer to the body of the paper and consider the tradeoffs discussed there in order to decide whether a given guideline is applicable to a particular application-specific component.

Guideline 1, page 37:

A *concept* should be represented as a single, generic package specification. All reusable components should be represented to the user in this way if possible. Even large subsystems should have a single point of visibility. Use subpackages within the abstraction to organize sets of related operations, if necessary, but maintain the "single top-level generic per component" mapping, even for components that are actually implemented using several packages. The user should be able to easily grasp the purpose/function of the abstraction, although it may take much more time to understand exactly how to fully utilize the supplied operations.

Guideline 2, page 37:

Each *concept* should provide one and only one abstraction—i.e., define a single object type. This will help to increase the understandability of the component, and also aid in separating pieces that may be independently reusable from one another.

Guideline 3, page 39:

There should be *no* fixed, horizontal coupling between a *concept* and other *concepts*. In other words, Ada packages that represent reusable component *concepts* should not **with** other packages. Instead, all definitions required to describe the *concept* should be passed in through generic parameters.

Guideline 4, page 47:

Each abstraction should be *robust*, meaning that it should provide a *complete* set of basic operations. The client can only access instances of an abstract type using the operations exported by the component's *concept*. Therefore, the operations provided should be sufficient for the reuser to construct any complex manipulations that are neede.l from them.

Guideline 5, page 47:

For the abstract types defined in a component, use **limited private**.

Guideline 6, page 47:

Always provide *initialize* and *finalize* operators for abstract types.

Guideline 7, page 47:

When writing a new component that uses other components, always faithfully apply the *initialize* and *finalize* operators. This guideline also applies to component reusers in general.

Guideline 8, page 48:

All abstract types in the *context* (i.e., which are generic parameters in the package specification) should be **limited private**. Similarly, *initialize* and *finalize* operations for such a type should also be part of the generic formal parameter list. These operations should be consistently applied within the component's body.

Guideline 9, page 48:

As a test of the robustness of both the generic parameters and the exported operations of a component that defines an abstract data type, consider ''composing'' the component with itself. For example, you should be able to create a ''stack or stacks'' simply by taking the exported type and operations from one stack instantiation and using them to instantiate the same generic again. There is not a general requirement for this capability, but it is nevertheless a useful way of testing the robustness of both the generic parameters and the exported operations.

Guideline 10, page 48:

To promote this composability and a uniform view of abstract types, *all* types should match the following minimum profile:

```
type Item is limited private;
procedure Swap(left, right : in out Item);
procedure Initialize(i : in out Item);
procedure Finalize(i : in out Item);
procedure Copy(from     : intem;
               into     : in out Item);
function Is_Equal(left, right : in Item) return boolean;
```

Figure 21. Minimum Operations for Generic Formal Type Parameters

Note that the duplication operation is called *copy* instead of *assign* to highlight the fact that it may be costly, rather than the fact that it can be used to move data. The names actually given to these operations is of secondary importance, however. It is the functionality provided by these operations, as well as the number and placement of arguments, that is important for composability.

Although *copy* and *is_equal* are not primitive operations, they are included because in the cases where they are needed, the extra cost of constructing them from the primitives without access to the underlying representation is often prohibitive, as discussed in Section 6.3. Need for the *swap* operator will be discussed in Section 5.2.2.

Guideline 11, page 52:

Each implementation of a *concept* should exist as a separate Ada generic package. However, all the package specifications for these implementations should be identical except for the package name. Also, these specifications may have additional generic parameters added that represent parameters to the corresponding implementation. These *implementation context* parameters, of course, are not necessarily uniform across all of the implementations. Thus, the Ada specifications may also differ in this respect.

Guideline 12, page 52:

The Ada package specifications for multiple implementations of a single concept should come from a common source, for example, using a preprocessor. The Ada package bodies for multiple implementations should share common code. Use lower-level generics (see [Musser89a] for an example), or a preprocessor so that common code comes from a single source.

Guideline 13, page 58:

Aliasing behavior (structural sharing) *is the responsibility of the abstraction, not the user!* "Structural sharing" semantics are often useful, but *all* basic operations must maintain the same semantics (in particular, *finalize*). Users should not be able to create aliases in an uncontrolled way (say, through use of the built in assignment operation). Instead, they may only call operations in the abstraction, which will then create aliases on their behalf (i.e., the package/abstraction must *always* maintain control over structural sharing).

Guideline 14, page 59:

Do not use the built in assignment operator as the basic data movement operator. Do not replace it with a *copy* operation. Instead, use a *swap* operation.

Guideline 15, page 60:

Every component should define a *swap* operation on its abstraction.

Guideline 16, page 60:

All operations Booch would classify as "constructors" or "selectors" should be designed using "swap" semantics, not "copy" semantics. The one name/one object paradigm Booch uses is the correct approach, although assignment/copy is the *wrong* underlying data movement primitive.

Guideline 17, page 60:

Provide *copy* and *is_equal* operators for all abstractions. Although these are really secondary operations, in the cases where they are needed, the additional costs of all the procedure calls involved in building one using primitive operations is unnecessary.

Guideline 18, page 88:

If possible, reusable components that export abstract data types should export a *complete* set of primary operations, so that the reuser can construct arbitrary iterations using these primary operations and the language mechanisms available for that purpose.

Guideline 19, page 88:

If a component designer chooses to export an explicit iterator, active, sequential access iterators are preferred over passive iterators. Such an iterator should protect objects of the abstract data type from structural modification while it is being iterated over. In addition, such an iterator should support destructive capabilities, and should also use explicit iterator state objects to ensure correct behavior under nested or concurrent iterations. The following profile is recommended:

```
type iterator_state is limited private;

procedure start_iteration(object      : in out ADT;
                          state       : in out iterator_state);
function being_iterated_over(object : in ADT) return boolean;
procedure access_current_entry(object     : in out ADT;
                               state      : in out iterator_state;
                               element    : in out element_type);
    -- "Swaps" the current value of ELEMENT with the value of the
    -- subpart of OBJECT located at the current position in the iteration STATE.
procedure advance(object      : in out ADT;
                  state       : in out iterator_state);
function iteration_is_complete(object     : in out ADT;
                               state      : in out iterator_state) return boolean;
procedure end_iteration(object      : in out ADT;
                        state       : in out iterator_state);

iterator_error : exception;
operation_not_allowed : exception;
```

Figure 36. Suggested Standard Profile for an Active, Sequential Iterator

Guideline 20, page 89:

If a component designer chooses to export an active, random access iterator, it should be provided *in addition to*, rather than in lieu of, an active, sequential access iterator. This will facilitate the development of tools that encapsulate iterative processes and that expect a common set of operations for performing such iterations. When a random access iterator is included, it should use explicit iterator state objects, support destructive iterations, and ensure correct behavior under both nested and concurrent iterations. The following profile is recommended:

```
type iterator_state is limited private;

procedure start_iteration(object     : in out ADT;
                          state      : in out iterator_state);
function being_iterated_over(object : in ADT) return boolean;
function size_of(object : in ADT) return integer;
procedure access_nth_entry(object       : in out ADT;
                           state        : in out iterator_state;
                           position     : in     integer;
                           element      : in out element_type);
     -- "Swaps" the current value of ELEMENT with the value of the
     -- subpart of OBJECT located at POSITION.  POSITION can be in the
     -- range 1 .. SIZE_OF(M).
procedure end_iteration(object : in out ADT;
                        state : in out iterator_state);

iterator_error : exception;
operation_not_allowed : exception;
```

Figure 37. Suggested Standard Profile for an Active, Random Access Iterator

Guideline 21, page 96:

If possible, components that support save and restore behavior for the abstractions they provide should follow the example in Figure 43 for operations exported by the component. Such components should also follow the example in Figure 42 for operations on generic formal type parameters.

APPENDIX C

SUMMARY OF ADA UNITS USED IN EXAMPLES

This Appendix presents complete versions of all the Ada packages introduced in this paper. Of particular importance is the specification for the Ada Package *GPD_pkg*, representing the GPD *concept* introduced in Section 3. For this package, first the original code is presented, followed by progressively more generalized versions, each incorporating the changes discussed in later sections of the paper. Each version is presented with ''change bars'' in the right hand margin to indicate where it differs from the version immediately preceding it in this Appendix. Other Ada units are also presented here.

1. "NAIVE" ADA SPECIFICATION FOR GPD *CONCEPT* (SECTION 3)

```
with text_io;
generic
        type Common_Node_Contents is private;
package GPD_pkg is

        type node_class is (gpd_empty,
                            gpd_integer,
                            gpd_boolean,
                            gpd_parent,
                            gpd_sequence);

        type gpd_type is private;
        null_gpd_node : constant gpd_type;

        ----------------------------------------------------------------
        -- The following 5 routines are common to all node classes.
        -- They include functions to determine the class of a node,
        -- deallocate a single node or a whole gpd structure, and
        -- read or write the COMMON_NODE_CONTENTS slot of any node.
        -- These 5 routines are followed by 5 subpackages, one for
        -- each node class. Each subpackage defines the node-class-specific
        -- functions for a give node-class. Note that some functions
        -- are overloaded (like NEW_NODE, etc.) if the desired node-class
        -- can be determined from the argument profile, but ambiguous cases
        -- (like NEW_NODE for generating a gpd_sequence vs. a gpd_empty)
        -- are given distinct names so they do not have to be qualified
        -- with subpackage names.
        --
        function node_class_of(node : in gpd_type) return node_class;
        procedure free(node : in out gpd_type);
                -- FREE is equivalent to recursively FREE'ing each child of
                -- a parent/sequence, then using FREE_SINGLE_NODE. Nodes are
                -- marked so that cycles in the GPD are handled correctly.
        procedure free_single_node(node : in out gpd_type);
                -- This routine frees the space occupied by a single node.
        ----------------------------------------------------------------
        -- All gpd nodes contain an element of type COMMON_NODE_CONTENTS.
        -- These functions allow access to this component of every node:
        --
        function get_data(node : in gpd_type) return common_node_contents;
        procedure put_data(node     : in out gpd_type;
                           data     : in     common_node_contents);
```

```
------------------------------------------------------------ ----
-- This subpackage defines the functions available for GPD nodes
-- of class GPD_EMPTY. A node of class GPD_EMPTY has no
-- outgoing nodes, and no slots other than one to hold
-- COMMON_NODE_CONTENTS.
--
package empty_node_pkg is
        function new_empty_node return gpd_type;
                -- Returns a node of class GPD_EMPTY, which can still
                -- hold COMMON_NODE_CONTENTS data.
end empty_node_pkg;

-----------------------------------------------------------------
-- This subpackage defines the functions available for GPD nodes
-- of class GPD_INTEGER. Each operation will ensure that its arg
-- is of class GPD_INTEGER, raising GPD_ERROR if otherwise.
--
package integer_node_pkg is
        function new_node(data : in integer) return gpd_type;
        function get_data(node : in gpd_type) return integer;
        procedure put_data(node     : in out gpd_type;
                           data     : in     integer);
end integer_node_pkg;

-----------------------------------------------------------------
-- This subpackage defines the functions available for GPD nodes
-- of class GPD_BOOLEAN. Each operation will ensure that its arg
-- is of class GPD_BOOLEAN, raising GPD_ERROR if otherwise.
--
package boolean_node_pkg is
        function new_node(data : in boolean) return gpd_type;
        function get_data(node : in gpd_type) return boolean;
        procedure put_data(node     : in out gpd_type;
                           data     : in     boolean);
end boolean_node_pkg;


-----------------------------------------------------------------
-- This subpackage defines the functions available for GPD nodes
-- of class GPD_PARENT. Each operation will ensure that its arg
-- is of class GPD_PARENT, raising GPD_ERROR if otherwise.
--
-- A node of class GPD_PARENT has an ordered list of children.
-- From the user's point of view, this list is organized as an array. The
-- length of this list is determined by the parameter to
-- MAKE_EMPTY_PARENT_NODE when the node was first created, and
-- this size cannot be changed for that parent node. The
-- children (some of which may be NULL_GPD_NODEs, the constant
-- defined earlier in the package for use as a null value) may be accessed
-- in any order using their positions relative to the beginning of
-- the list (i.e., their array index). Indices run from 1 to
-- MAX_CHILDREN.
--
package parent_node_pkg is
        function make_empty_parent_node(
                max_children : in positive := 2) return gpd_type;
        function max_children(node : in gpd_type) return natural;
        procedure put_child(child_node       : in     gpd_type;
                            parent_node      : in out gpd_type;
                            position         : in     positive);
                -- This routine assigns the specified CHILD_NODE into
                -- the specified position of the PARENT_NODE's conceptual
                -- array of outgoing links. This overwrites any previous value
                -- there. Since objects of GPD_TYPE are represented as pointer
                -- values, this introduces structural sharing.
        function get_child(parent_node       : in gpd_type;
                           position          : in positive) return gpd_type;
end parent_node_pkg;

-----------------------------------------------------------------
-- This subpackage defines the functions available for GPD nodes
-- of class GPD_SEQUENCE. Each operation will ensure that its arg
-- is of class GPD_SEQUENCE, raising GPD_ERROR if otherwise.
--
-- A sequence node contains an arbitrarily long list of child
-- nodes, which may themselves be other sequences. These children
-- can be accessed, and the list of children modified, by the
-- subroutines in this package.
--
package sequence_node_pkg is
        subtype sequence_type is gpd_type;
                -- This subtype is just used for clarity in the
                -- declarations below to show where a node of class
                -- GPD_SEQUENCE is expected. If a node of a different
                -- class is used where this subtype appears, GPD_ERROR
                -- will be raised.
        function make_empty_sequence_node return gpd_type;
                -- Return a new GPD_SEQUENCE node with no outgoing links.
        procedure append(seq                 : in out sequence_type;
                         new_element         : in     gpd_type);
        procedure remove_head(seq      : in out sequence_type;
                              head     :    out gpd_type);
        procedure prepend(seq                : in out sequence_type;
                          new_element        : in     gpd_type);
        procedure remove_tail(seq        : in out sequence_type;
                              tail       :    out gpd_type);
        procedure read_and_consume(seq              : in out sequence_type;
                                   element          :    out gpd_type;
                                   N                : in     positive := 1);
                -- Removes the Nth element of the list of outgoing links,
                -- placing its value in ELEMENT.
        procedure read_nth_element(seq              : in out sequence_type;
                                   element          :    out gpd_type;
                                   N                : in     positive := 1);
                -- places the value of the Nth element of the list of outgoing
                -- links in ELEMENT without altering the list.
        procedure consume(seq      : in out sequence_type;
                          N        : in     positive := 1);
                -- Removes the Nth element of the list, without calling
                -- FREE on the contents. The reference stored in that
                -- outgoing link is lost.
        procedure consume_n_elements(seq      : in out sequence_type;
                                     N        : in     positive);
                -- Removes the first N elements of the list, without calling
                -- FREE on any of the contents. The references stored in those
                -- outgoing links are lost.
        function length(seq : in sequence_type) return natural;
                -- returns the number of outgoing links
        procedure reverse_sequence(seq : in out sequence_type);
                -- reverses the order of the list of outgoing links
        function copy(sequence : in sequence_type) return sequence_type;
                -- produces a new node of class GPD_SEQUENCE with an
                -- identical list of outgoing links
        procedure concat(onto, from : in out sequence_type);
                -- remove all outgoing links from ONTO, concatenating them
                -- onto FROM's list of outgoing links. At completion,
                -- ONTO will have an empty list of links.
        function is_empty(seq : in sequence_type) return boolean;
                -- are there any outgoing links from SEQ?

end sequence_node_pkg;


-------------------------------------------------------------------
-- Errors:
-- This package only defines one exception, GPD_ERROR. This
-- exception is raised whenever a node-class-specific function
-- or procedure is called with an argument of the wrong class.
-- The exception CONSTRAINT_ERROR is raised if NULL_GPD_NODE
-- is passed into a routine.
--
```

```
        gpd_error : exception;

private
        type gpd_block(class              : node_class        := gpd_empty;
                       top_size           : natural           := 0;
                       bottom_size        : natural           := 0);
        type gpd_type is access gpd_block;
        null_gpd_node : constant gpd_type := null;

end GPD_pkg;
```

2. GPD *CONCEPT* SPECIFICATION, AS SUGGESTED IN SECTION 5.1

```
with text_io;
generic
        type Common_Node_Contents is limited private;                           |
        with procedure initialize(data : in out Common_Node_Contents);          |
        with procedure finalize(data : in out Common_Node_Contents);            |
        with procedure assign(from : in      Common_Node_Contents;              |
                              into  : in out Common_Node_Contents);             |
                -- ASSIGN should FINALIZE INTO first, then copy the value       |
                -- of FROM.                                                     |
package GPD_pkg is

        type node_class is (gpd_empty,
                            gpd_integer,
                            gpd_boolean,
                            gpd_parent,
                            gpd_sequence);

        type gpd_type is limited private;                                       |
                -- The basic operations for this type:                          |
        procedure initialize(node : in out gpd_type);                           |
        procedure finalize(node              : in out gpd_type;                 |
                           recurse           : in     boolean     := true;      |
                           finalize_cnc      : in     boolean     := true);     |
                -- If RECURSE is true, FINALIZE is called "recursively" (cycles |
                -- are detected and handled correctly) on all links going out   |
                -- from NODE first.  If FINALIZE_CNC is true, the formal        |
                -- parameter FINALIZE is called on the Common_Node_Contents of NODE. |
                -- Finally, storage for NODE is reclaimed, and NODE is given    |
                -- the initial value, NULL_GPD_NODE.  Finalize thus replaces    |
                -- both the FREE and FREE_SINGLE_NODE procedures from the       |
                -- earlier package.                                             |
        procedure assign(from   : in     gpd_type;                              |
                         into   : in out gpd_type);                             |
                -- ASSIGN FINALIZEs INTO first, then creates an identical       |
                -- copy of FROM (including recursively copying all nodes pointed |
                -- to by outgoing links--cycles are detected, and replicated    |
                -- appropriately).                                              |
null_gpd_node : constant gpd_type; -- the initial value for this type

-------------------------------------------------------------------
-- The following 3 routines are common to all node classes.
-- They include functions to determine the class of a node and
-- read or write the Common_Node_Contents slot of any node.
-- These 3 routines are followed by 5 subpackages, one for
-- each node class. Each subpackage defines the node-class-specific
-- functions for a give node-class. Note that some functions
-- are overloaded (like "new_node", etc.) if the desired node-class
-- can be determined from the argument profile, but ambiguous cases
-- (like "new_node" for generating a gpd_sequence vs. a gpd_empty)
-- are given distinct names so they do not have to be qualified
-- with subpackage names.
--
function node_class_of(node : in gpd_type) return node_class;
procedure get_data(node     : in     gpd_type;
                   data     : in out Common_Node_Contents);
procedure put_data(node     : in out gpd_type;
                   data     : in     Common_Node_Contents);



-------------------------------------------------------------------
-- This subpackage defines the functions available for GPD nodes
-- of class GPD_EMPTY. A node of class GPD_EMPTY has no
-- outgoing nodes, and no slots other than one to hold
-- COMMON_NODE_CONTENTS.
--
package empty_node_pkg is
      procedure new_empty_node(node : in out gpd_type);
            -- Makes sure NODE is finalized, then replaces it
            -- with a node of class "gpd_empty", which can still
            -- hold Common_Node_Contents data.
end empty_node_pkg;

-------------------------------------------------------------------
-- This subpackage defines the functions available for gpd nodes
-- of class "gpd_integer." Each operation will ensure that its arg
-- is of class "gpd_integer," raising GPD_ERROR if otherwise.
--
package integer_node_pkg is
      procedure new_node(data     : in     integer;
                         node     : in out gpd_type);
      function get_data(node : in gpd_type) return integer;
      procedure put_data(node     : in out gpd_type;
                         data     : in     integer);
end integer_node_pkg;
```

```
--------------------------------------------------------------
-- This subpackage defines the functions available for gpd nodes
-- of class "gpd_boolean." Each operation will ensure that its arg
-- is of class "gpd_boolean," raising GPD_ERROR if otherwise.
--
package boolean_node_pkg is
        procedure new_node(data      : in      boolean;
                           node      : in out gpd_type);
        function get_data(node : in gpd_type) return boolean;
        procedure put_data(node     : in out gpd_type;
                           data     : in      boolean);
end boolean_node_pkg;


--------------------------------------------------------------
-- This subpackage defines the functions available for GPD nodes
-- of class GPD_PARENT. Each operation will ensure that its arg
-- is of class GPD_PARENT, raising GPD_ERROR if otherwise.
--
-- A node of class GPD_PARENT has an ordered list of children.
-- From the user's point of view, this list is organized as an array. The
-- length of this list is determined by the parameter to
-- MAKE_EMPTY_PARENT_NODE when the node was first created, and
-- this size cannot be changed for that parent node. The
-- children (some of which may be NULL_GPD_NODEs, the constant
-- defined earlier in the package for use as a null value) may be accessed
-- in any order using their positions relative to the beginning of
-- the list (i.e., their array index). Indices run from 1 to
-- MAX_CHILDREN.
--
package parent_node_pkg is
        procedure make_empty_parent_node(n : in positive := 2;
                                         node : in out gpd_type);
        function max_children(node : in gpd_type) return natural;
        procedure put_child(child_node      : in      gpd_type;
                            parent_node     : in out gpd_type;
                            position        : in      positive);
        procedure get_child(parent_node     : in gpd_type;
                            position        : in positive;
                            child           : in out gpd_type);
end parent_node_pkg;

--------------------------------------------------------------
-- This subpackage defines the functions available for gpd nodes
-- of class "gpd_sequence." Each operation will ensure that its arg
-- is of class "gpd_sequence," raising GPD_ERROR if otherwise.
--
```

```
-- A sequence node contains an arbitrarily long list of child
-- nodes, which may themselves be other sequences. These children
-- can be accessed, and the list of children modified, by the
-- subroutines in this package.
--
package sequence_node_pkg is
        subtype sequence_type is gpd_type;
                -- This subtype is just used for clarity in the
                -- declarations below to show where a node of class
                -- "gpd_sequence" is expected. If a node of a different
                -- class is used where this subtype appears, GPD_ERROR
                -- will be raised.
        procedure make_empty_sequence_node(node : in out gpd_type);
                -- Return a new gpd_sequence node with no outgoing links.
        procedure append(seq                    : in out sequence_type;
                         new_element            : in     gpd_type);
        procedure remove_head(seq       : in out sequence_type;
                              head      :    out gpd_type);
        procedure prepend(seq                   : in out sequence_type;
                          new_element           : in     gpd_type);
        procedure remove_tail(seq       : in out sequence_type;
                              tail      :    out gpd_type);
        procedure read_and_consume(seq          : in out sequence_type;
                                   element      :    out gpd_type;
                                   N            : in     positive := 1);
                -- Removes the Nth element of the list of outgoing links,
                -- placing its value in ELEMENT.
        procedure read_nth_element(seq          : in out sequence_type;
                                   element      :    out gpd_type;
                                   N            : in     positive := 1);
                -- places the value of the Nth element of the list of outgoing
                -- links in ELEMENT without altering the list.
        procedure consume(seq                   : in out sequence_type;
                          N                     : in     positive      := 1;
                          finalize              : in     boolean       := true;
                          recurse               : in     boolean       := true;
                          finalize_cnc          : in     boolean       := true);
                -- Removes the Nth element of the list. If FINALIZE is true,
                -- the FINALIZE operation is invoked on the value of the
                -- Nth element before it is removed. The parameters s-1RECURSE
                -- and FINALIZE_CNC are passed to the FINALIZE operation
                -- if it is invoked.
       procedure consume_n_elements(seq         : in out sequence_type;
                                    N           : in     positive;
                                    finalize    : in     boolean     := true;
                                    recurse     : in     boolean     := true;
                                    finalize_cnc : in    boolean     := true);
```

```
          -- Remove the first N elements of the list. See CONSUME for
          -- an explanation of the final three arguments.
      function length(seq : in sequence_type) return natural;
          -- returns the number of outgoing links
      procedure reverse_sequence(seq : in out sequence_type);
          -- reverses the order of the list of outgoing links
      procedure copy(original     : in     sequence_type;
                     duplicate    : in out sequence_type);
          -- produces a new node of class "gpd_sequence" with an
          -- identical list of outgoing links
      procedure concat(onto, from : in out sequence_type);
          -- remove all outgoing links from ONTO, concatenating them
          -- onto FROM's list of outgoing links. At completion,
          -- ONTO will have an empty list of links.
      function is_empty(seq : in sequence_type) return boolean;
          -- are there any outgoing links from SEQ?

   end sequence_node_pkg;


   ------------------------------------------------------------------
   -- Errors:
   -- This package only defines one exception, GPD_ERROR. This
   -- exception is raised whenever a node-class-specific function
   -- or procedure is called with an argument of the wrong class.
   -- The exception CONSTRAINT_ERROR is raised if NULL_GPD_NODE
   -- is passed into a routine.
   --
   gpd_error       : exception;

private
   type gpd_block(class         : node_class      := gpd_empty;
                  top_size      : natural         := 0;
                  bottom_size   : natural         := 0);
   type gpd_type is access gpd_block;
   null_gpd_node : constant gpd_type := null;

end GPD_pkg;
```

3. GPD *CONCEPT* SPECIFICATION, AS SUGGESTED IN SECTION 5.2.2

```
with text_io, basic_type_support;
generic
        type Common_Node_Contents is limited private;
        with procedure initialize(data : in out Common_Node_Contents);
        with procedure finalize(data : in out Common_Node_Contents);
        with procedure swap(left     : in out Common_Node_Contents;
                            right    : in out Common_Node_Contents);
package GPD_pkg is

        -- For use in single-threaded applications where concurrency
        -- protection for memory management is NOT necessary.

        type node_class is (gpd_empty,
                            gpd_integer,
                            gpd_boolean,
                            gpd_parent,
                            gpd_sequence);
                -- The type NODE_CLASS is left unencapsulated for simplicity

        type gpd_type is limited private;
                -- The basic operations for this type:
        procedure initialize(node : in out gpd_type);
                -- ensures that NODE = NULL_GPD_NODE
                -- (NULL_GPD_NODE is the initial value for all elements of this
                -- type).
        procedure finalize(data : in out gpd_type);
                -- If NODE is referenced from multiple locations, this reference
                -- is set to NULL_GPD_NODE. If this is the last remaining reference
                -- to NODE, then storage for NODE is reclaimed, and NODE is given
                -- the initial value, NULL_GPD_NODE.
        procedure swap(left      : in out gpd_type;
                       right     : in out gpd_type);
                -- exchanges the contents of left and right
        procedure duplicate(original     : in     gpd_type;
                            copy         : in out gpd_type);
                -- DUPLICATE FINALIZE COPY first, then creates an identical
                -- copy of the ORIGINAL (including recursively copying all
                -- nodes pointed to by outgoing links--cycles are detected,
        -- and replicated appropriately).

null_gpd_node : constant gpd_type; -- the initial value for this type

-------------------------------------------------------------------
-- The following 2 routines are common to all node classes.
-- They include functions to determine the class of a node and
-- to access the Common_Node_Contents slot of any node.
-- These functions are followed by 5 subpackages, one for
-- each node class. Each subpackage defines the node-class-specific
-- functions and procedures for a give node-class. Note that some
-- routines are overloaded (like "new_node", etc.) if the
-- desired node-class can be determined from the argument profile,
-- but ambiguous cases (like "new_node" for generating a
-- gpd_sequence vs. a gpd_empty) are given distinct names so
-- they do not have to be qualified with subpackage names.
--
function node_class_of(node : in gpd_type) return node_class;

-- The FREE routines in the previous versions are replaced by
-- the FINALIZE routine above.

procedure access_contents(node     : in out gpd_type;
                          data     : in out Common_Node_Contents);
        -- Exchanges the value (via a call to the appropriate SWAP)
        -- of DATA with the value of the Common_Node_Contents slot of
        -- NODE.


-------------------------------------------------------------------
-- This subpackage defines the functions available for gpd nodes
-- of class "gpd_empty." Each operation will ensure that its arg
-- is of class "gpd_empty," raising GPD_ERROR if otherwise.
--
-- A node of class "gpd_empty" has no outgoing nodes, and
-- no slots other than one to hold Common_Node_Contents.
--
package empty_node_pkg is
        procedure new_empty_node(node : in out gpd_type);
                -- Makes sure NODE is finalized, then replaces it
                -- with a node of class "gpd_empty", which can still
                -- hold Common_Node_Contents data.
end empty_node_pkg;


-- The routines in the following 2 packages use the SWAP
-- routines from the package BASIC_TYPE_SUPPORT to "access"
-- the internal contents of each node-class (gpd_integer and
-- gpd_boolean). This isn't strictly necessary for these
-- types (which may not be strongly encapsulated), but does
-- serve as an example of how strongly encapsulated types
-- would be treated.
--
-- In each case, an "access" routine replaces both the GET_DATA
-- and PUT_DATA routines found in other versions of this package.
-- If duplication of internal data were required by the user,
-- he could use the appropriate COPY or DUPLICATE routines
-- exported by BASIC_TYPE_SUPPORT (or whatever package defined
-- the type under consideration).

----------------------------------------------------------------
-- This subpackage defines the functions available for gpd nodes
-- of class "gpd_integer." Each operation will ensure that its arg
-- is of class "gpd_integer," raising GPD_ERROR if otherwise.
--
package integer_node_pkg is
        procedure new_node(data     : in      integer;
                           node     : in out gpd_type);
        procedure access_data(node   : in out gpd_type;
                              data   : in out integer);
end integer_node_pkg;

----------------------------------------------------------------
-- This subpackage defines the functions available for gpd nodes
-- of class "gpd_boolean." Each operation will ensure that its arg
-- is of class "gpd_boolean," raising GPD_ERROR if otherwise.
--
package boolean_node_pkg is
        procedure new_node(data     : in      boolean;
                           node     : in out gpd_type);
        procedure access_data(node   : in out gpd_type;
                              data   : in out boolean);
end boolean_node_pkg;


----------------------------------------------------------------
-- This subpackage defines the functions available for gpd nodes
-- of class "gpd_parent." Each operation will ensure that its arg
-- is of class "gpd_parent," raising GPD_ERROR if otherwise.
--
-- A node of class "gpd_parent" has an ordered list of children. The
-- length of this list is determined by the parameter to
-- Make_Empty_Parent_Node when the node was first created. The
-- children (some of which may be Null_GPD_Nodes) may be accessed
-- in any order by their relative positiions frum the beginning of
-- the list.
--
package parent_node_pkg is
        procedure make_empty_parent_node(
                n          : in      positive := 2;
                node       : in out gpd_type);
        function max_children(node : in gpd_type) return natural;
                -- This routine remains a function, and relies on the
                -- COPY routine exported by BASIC_TYPE_SUPPORT. The user
                -- can't "access" the number of children--it must be read-only
                -- from his point of view.
        procedure access_child(
                child_node          : in out gpd_type;
                parent_node         : in out gpd_type;
                position            : in      positive);
                -- Remember, this _replaces_ the value of the Nth
                -- outgoing link with the value of CHILD_NODE, and
                -- replaces the value of CHILD_NODE with the value of
                -- the Nth outgoing link (a "swap").
        procedure put_child(
                child_node          : in      gpd_type;
                parent_node         : in out gpd_type;
                position            : in      positive);
                -- Rather than "swapping", this does what you would
                -- expect: it "links" the Nth outgoing link of the
                -- PARENT_NODE
                -- to the CHILD_NODE. This is a "restricted" form of
                -- aliasing, which is completely under the control of
                -- this module (i.e., not visible or accessible to the
                -- end user).
end parent_node_pkg;

-----------------------------------------------------------------
-- This subpackage defines the functions available for gpd nodes
-- of class "gpd_sequence." Each operation will ensure that its arg
-- is of class "gpd_sequence," raising GPD_ERROR if otherwise.
--
-- A sequence node contains an arbitrarily long list of child
-- nodes, which may themselves be other sequences. These children
-- can be accessed, and the list of children modified, by the
-- subroutines in this package.
--
package sequence_node_pkg is
        subtype sequence_type is gpd_type;
                -- This subtype is just used for clarity in the
        -- declarations below to show where a node of class
        -- "gpd_sequence" is expected. If a node of a different
        -- class is used where this subtype appears, GPD_ERROR
        -- will be raised.
procedure make_empty_sequence_node(node : in out gpd_type);
        -- Create a new gpd_sequence node with no outgoing links.


-- Rather than "swapping", these operations introduces structural
-- sharing semantics. They "link" the corresponding outgoing link
-- of the SEQUENCE_NODE to the CHILD_NODE given in the
-- argument (or remove such a link). This is a "restricted" form
-- of aliasing, which is completely under the control of this
-- module (i.e., not accessible to the end user).
--
procedure append(
        seq                 : in out sequence_type;
        new_element         : in     gpd_type);
        -- adds a new outgoing link to the end of SEQ's
        -- list of links, then places a reference to the NEW_ELEMENT
        -- in this outgoing link.
procedure remove_head(
        seq     : in out sequence_type;
        head    : in out gpd_type);
        -- removes the first outgoing link on SEQ's list, and
        -- returns the object pointed to by that link.
procedure prepend(
        seq                 : in out sequence_type;
        new_element         : in     gpd_type);
        -- like APPEND, but for the beginning of the list.
procedure remove_tail(
        seq     : in out sequence_type;
        tail    :    out gpd_type);
        -- like REMOVE_HEAD, but for the end of the list.
procedure access_nth_element(
        seq          : in out sequence_type;
        element      : in out gpd_type;
        N            : in     positive := 1);
        -- "Swaps" the node pointed to by the Nth outgoing link with
        -- the current value of ELEMENT.
procedure consume(
        seq     : in out sequence_type;
        N       : in     positive := 1);
        -- Removes the Nth outgoing link from the list. FINALIZE is called
        -- on the contents before the link is removed.
procedure consume_n_elements(
        seq     : in out sequence_type;
```

```
                N       : in        positive);                                  |
                -- Removes the first N outgoing links from the list. FINALIZE is   |
                -- called on the contents of each link before it is removed.      |
        function length(seq : in sequence_type) return natural;
                -- Returns the number of outgoing links.                           |
        procedure reverse_sequence(seq : in out sequence_type);
                -- Reverses the order of the list of outgoing links.               |
        procedure copy(                                                            |
                original        : in        sequence_type;                         |
                duplicate       : in out sequence_type);                           |
                -- Produces a new node of class "gpd_sequence" with an             |
                -- identical list of outgoing links. Unlike the DUPLICATE          |
                -- operation, however, both the ORIGINAL and the DUPLICATE conceptually |
                -- share structural references to the same children (DUPLICATE     |
                -- would create a new set of identical children).                  |
        procedure concat(                                                          |
                onto    : in out sequence_type;                                    |
                from    : in out sequence_type);                                   |
                -- Removes all outgoing links from ONTO, concatenating them        |
                -- onto FROM's list of outgoing links. At completion,
                -- ONTO will have an empty list of links.
        function is_empty(seq : in sequence_type) return boolean;
                -- Are there any outgoing links from SEQ?                          |

    end sequence_node_pkg;


    ------------------------------------------------------------------
    -- Errors:
    -- This package only defines one exception, GPD_ERROR. This
    -- exception is raised whenever a node-class-specific function
    -- or procedure is called with an argument of the wrong class.
    -- The exception CONSTRAINT_ERROR is raised if NULL_GPD_NODE
    -- is passed into a routine.
    --
    gpd_error           : exception;

private
    type gpd_block(class                : node_class        := gpd_empty;
                   top_size             : natural           := 0;
                   bottom_size          : natural           := 0);
    type gpd_type is access gpd_block;
    null_gpd_node : constant gpd_type := null;

end GPD_pkg;
```

4. GPD *CONCEPT* SPECIFICATION, AS SUGGESTED IN SECTION 5.2.3

```
with text_io, basic_type_support;
generic
        type Common_Node_Contents is limited private;
        with procedure initialize(data : in out Common_Node_Contents);
        with procedure finalize(data : in out Common_Node_Contents);
        with procedure swap(left        : in out Common_Node_Contents;
                            right       : in out Common_Node_Contents);
        with procedure read(file        : in     text_io.file_type;
                            data        : in out Common_Node_Contents);
                -- The DATA parameter of READ is mode in out because
                -- READ FINALIZEs the incoming value of DATA
                -- before placing the result of the READ operation in it.
        with procedure write(file       : in text_io.file_type;
                             data       : in Common_Node_Contents);
                -- In effect, WRITE sends a "copy" of DATA to FILE.
package GPD_pkg is

        -- For use in single-threaded applications where concurrency
        -- protection for memory management is NOT necessary.

        type node_class is (gpd_empty,
                            gpd_integer,
                            gpd_boolean,
                            gpd_parent,
                            gpd_sequence,
                            gpd_user_defined);
                -- The type NODE_CLASS is left unencapsulated for simplicity

        type gpd_type is limited private;
                -- The basic operations for this type:
        procedure initialize(node : in out gpd_type);
                -- ensures that NODE = NULL_GPD_NODE
                -- (NULL_GPD_NODE is the initial value for all elements of this
                -- type).
        procedure finalize(data : in out gpd_type);
                -- If NODE is referenced from multiple locations, this reference
                -- is set to NULL_GPD_NODE. If this is the last remaining reference
                -- to NODE, then storage for NODE is reclaimed, and NODE is given
        -- the initial value, NULL_GPD_NODE.
procedure swap(left         : in out gpd_type;
               right        : in out gpd_type);
        -- exchanges the contents of left and right
procedure duplicate(original      : in      gpd_type;
                    copy          : in out gpd_type);
        -- DUPLICATE FINALIZE COPY first, then creates an identical
        -- copy of the ORIGINAL (including recursively copying all
        -- nodes pointed to by outgoing links--cycles are detected,
        -- and replicated appropriately).

null_gpd_node : constant gpd_type;  -- the initial value for this type

-----------------------------------------------------------------
-- The following 4 routines are common to all node classes.
-- They include functions to determine the class of a node,
-- to access the Common_Node_Contents slot of any node, and to save
-- or restore a GPD node to or from a file.
-- These functions are followed by 5 subpackages, one for
-- each node class.  Each subpackage defines the node-class-specific
-- functions and procedures for a give node-class.  Note that some
-- routines are overloaded (like "new_node", etc.) if the
-- desired node-class can be determined from the argument profile,
-- but ambiguous cases (like "new_node" for generating a
-- gpd_sequence vs. a gpd_empty) are given distinct names so
-- they do not have to be qualified with subpackage names.
--
function node_class_of(node : in gpd_type) return node_class;

-- The FREE routines in the previous versions are replaced by
-- the FINALIZE routine above.

procedure access_contents(node       : in out gpd_type;
                          data       : in out Common_Node_Contents);
        -- Exchanges the value (via a call to the appropriate SWAP)
        -- of DATA with the value of the Common_Node_Contents slot of
        -- NODE.

procedure save(file       : in text_io.file_type;
               gpd        : in gpd_type);
        -- In effect, a "copy" of GPD is placed on the FILE.

procedure restore(file       : in      text_io.file_type;
                  gpd        : in out gpd_type);
        -- The GPD parameter of RESTORE is mode in out so that its prcvious
        -- value can be finalized before the new structure is assigned to it.
```

```
-----------------------------------------------------------------
-- This subpackage defines the functions available for gpd nodes
-- of class "gpd_empty." Each operation will ensure that its arg
-- is of class "gpd_empty," raising GPD_ERROR if otherwise.
--
-- A node of class "gpd_empty" has no outgoing nodes, and
-- no slots other than one to hold Common_Node_Contents.
--
package empty_node_pkg is
        procedure new_empty_node(node : in out gpd_type);
                -- Makes sure NODE is finalized, then replaces it
                -- with a node of class "gpd_empty", which can still
                -- hold Common_Node_Contents data.
end empty_node_pkg;



-- The routines in the following 2 packages use the SWAP
-- routines from the package BASIC_TYPE_SUPPORT to "access"
-- the internal contents of each node-class (gpd_integer and
-- gpd_boolean). This isn't strictly necessary for these
-- types (which may not be strongly encapsulated), but does
-- serve as an example of how strongly encapsulated types
-- would be treated.
--
-- In each case, an "access" routine replaces both the GET_DATA
-- and PUT_DATA routines found in other versions of this package.
-- If duplication of internal data were required by the user,
-- he could use the appropriate COPY or DUPLICATE routines
-- exported by BASIC_TYPE_SUPPORT (or whatever package defined
-- the type under consideration).

-----------------------------------------------------------------
-- This subpackage defines the functions available for gpd nodes
-- of class "gpd_integer." Each operation will ensure that its arg
-- is of class "gpd_integer," raising GPD_ERROR if otherwise.
--
package integer_node_pkg is
        procedure new_node(data      : in      integer;
                           node      : in out gpd_type);
        procedure access_data(node    : in out gpd_type;
                              data    : in out integer);
end integer_node_pkg;

-----------------------------------------------------------------
-- This subpackage defines the functions available for gpd nodes
-- of class "gpd_boolean." Each operation will ensure that its arg
-- is of class "gpd_boolean," raising GPD_ERROR if otherwise.
--
package boolean_node_pkg is
        procedure new_node(data        : in      boolean;
                           node        : in out gpd_type);
        procedure access_data(node      : in out gpd_type;
                              data      : in out boolean);
end boolean_node_pkg;


-----------------------------------------------------------------
-- This subpackage defines the functions available for gpd nodes
-- of class "gpd_parent." Each operation will ensure that its arg
-- is of class "gpd_parent," raising GPD_ERROR if otherwise.
--
-- A node of class "gpd_parent" has an ordered list of children. The
-- length of this list is determined by the parameter to
-- Make_Empty_Parent_Node when the node was first created. The
-- children (some of which may be Null_GPD_Nodes) may be accessed
-- in any order by their relative positiions from the beginning of
-- the list.
--
package parent_node_pkg is
        procedure make_empty_parent_node(
                n          : in      positive := 2;
                node       : in out gpd_type);
        function max_children(node : in gpd_type) return natural;
                -- This routine remains a function, and relies on the
                -- COPY routine exported by BASIC_TYPE_SUPPORT. The user
                -- can't "access" the number of children--it must be read-only
                -- from his point of view.
        procedure access_child(
                child_node              : in out gpd_type;
                parent_node             : in out gpd_type;
                position                : in      positive);
                -- Remember, this _replaces_ the value of the Nth
                -- outgoing link with the value of CHILD_NODE, and
                -- replaces the value of CHILD_NODE with the value of
                -- the Nth outgoing link (a "swap").
        procedure put_child(
                child_node              : in      gpd_type;
                parent_node             : in out gpd_type;
                position                : in      positive);
                -- Rather than "swapping", this does what you would
                -- expect: it "links" the Nth outgoing link of the
                -- PARENT_NODE
                -- to the CHILD_NODE. This is a "restricted" form of
                -- aliasing, which is completely under the control of
                -- this module (i.e., not visible or accessible to the
                -- end user).
end parent_node_pkg;

------------------------------------------------------------------
-- This subpackage defines the functions available for gpd nodes
-- of class "gpd_sequence." Each operation will ensure that its arg
-- is of class "gpd_sequence," raising GPD_ERROR if otherwise.
--
-- A sequence node contains an arbitrarily long list of child
-- nodes, which may themselves be other sequences. These children
-- can be accessed, and the list of children modified, by the
-- subroutines in this package.
--
package sequence_node_pkg is
        subtype sequence_type is gpd_type;
                -- This subtype is just used for clarity in the
                -- declarations below to show where a node of class
                -- "gpd_sequence" is expected. If a node of a different
                -- class is used where this subtype appears, GPD_ERROR
                -- will be raised.
        procedure make_empty_sequence_node(node : in out gpd_type);
                -- Create a new gpd_sequence node with no outgoing links.


        -- Rather than "swapping", these operations introduces structural
        -- sharing semantics. They "link" the corresponding outgoing link
        -- of the SEQUENCE_NODE to the CHILD_NODE given in the
        -- argument (or remove such a link). This is a "restricted" form
        -- of aliasing, which is completely under the control of this
        -- module (i.e., not accessible to the end user).
        --
        procedure append(
                seq               : in out sequence_type;
                new_element       : in     gpd_type);
                -- adds a new outgoing link to the end of SEQ's
                -- list of links, then places a reference to the NEW_ELEMENT
                -- in this outgoing link.
        procedure remove_head(
                seq     : in out sequence_type;
                head    : in out gpd_type);
                -- removes the first outgoing link on SEQ's list, and
                -- returns the object pointed to by that link.
        procedure prepend(
                seq               : in out sequence_type;
                new_element       : in     gpd_type);
        -- like APPEND, but for the beginning of the list.
    procedure remove_tail(
        seq     : in out sequence_type;
        tail    :    out gpd_type);
        -- like REMOVE_HEAD, but for the end of the list.
    procedure access_nth_element(
        seq         : in out sequence_type;
        element     : in out gpd_type;
        N           : in     positive := 1);
        -- "Swaps" the node pointed to by the Nth outgoing link with
        -- the current value of ELEMENT.
    procedure consume(
        seq     : in out sequence_type;
        N       : in     positive := 1);
        -- Removes the Nth outgoing link from the list. FINALIZE is called
        -- on the contents before the link is removed.
    procedure consume_n_elements(
        seq     : in out sequence_type;
        N       : in     positive);
        -- Removes the first N outgoing links from the list. FINALIZE is
        -- called on the contents of each link before it is removed.
    function length(seq : in sequence_type) return natural;
        -- Returns the number of outgoing links.
    procedure reverse_sequence(seq : in out sequence_type);
        -- Reverses the order of the list of outgoing links.
    procedure copy(
        original        : in     sequence_type;
        duplicate       : in out sequence_type);
        -- Produces a new node of class "gpd_sequence" with an
        -- identical list of outgoing links. Unlike the DUPLICATE
        -- operation, however, both the ORIGINAL and the DUPLICATE conceptually
        -- share structural references to the same children (DUPLICATE
        -- would create a new set of identical children).
    procedure concat(
        onto    : in out sequence_type;
        from    : in out sequence_type);
        -- Removes all outgoing links from ONTO, concatenating them
        -- onto FROM's list of outgoing links. At completion,
        -- ONTO will have an empty list of links.
    function is_empty(seq : in sequence_type) return boolean;
        -- Are there any outgoing links from SEQ?

end sequence_node_pkg;

-----------------------------------------------------------------          |
-- This subpackage defines the functions available for gpd nodes          |
-- of class "gpd_user_defined." This generic subpackage allows the        |
      -- user to create GPD nodes that can contain any user-defined type.
      -- Each operation will ensure that its arg is of class "gpd_user_defined,"
      -- as well as ensuring that it contains data of the correct user-defined
      -- type. The exception GPD_ERROR will be raised otherwise.
      --
      generic
            type user_defined_data is limited private;
            with procedure initialize(data : in out user_defined_data);
            with procedure finalize(data : in out user_defined_data);
            with procedure swap(left          : in out user_defined_data;
                                right         : in out user_defined_data);
            with procedure read(file       : in      text_io.file_type;
                                data       : in out user_defined_data);
                  -- The DATA parameter of READ is mode in out because
                  -- READ FINALIZEs the incoming value of DATA
                  -- before placing the result of the READ operation in it.
            with procedure write(file       : in text_io.file_type;
                                 data       : in user_defined_data);
                  -- In effect, WRITE sends a "copy" of DATA to FILE.
      package user_defined_node_pkg is

            procedure new_node(data       : in out user_defined_data;
                               node       : in out gpd_type);
            procedure access_data(node       : in out gpd_type;
                                  data       : in out user_defined_data);

      end user_defined_node_pkg;

      ------------------------------------------------------------------
      -- Errors:
      -- This package only defines one exception, GPD_ERROR. This
      -- exception is raised whenever a node class-specific function
      -- or procedure is called with an argument of the wrong class.
      -- The exception CONSTRAINT_ERROR is raised if NULL_GPD_NODE
      -- is passed into a routine.
      --
      gpd_error          : exception;

private
      type gpd_block(class              : node_class          := gpd_empty;
                     top_size           : natural             := 0;
                     bottom_size        : natural             := 0);
      type gpd_type is access gpd_block;
      null_gpd_node : constant gpd_type := null;

end GPD_pkg;
```

5. GPD *CONCEPT* SPECIFICATION, AS SUGGESTED IN SECTION 6.4

```
with text_io, basic_type_support;
generic
        type Common_Node_Contents is limited private;
        with procedure initialize(data : in out Common_Node_Contents);
        with procedure finalize(data : in out Common_Node_Contents);
        with procedure swap(left        : in out Common_Node_Contents;
                            right       : in out Common_Node_Contents);

        -- For saving and restoring a single object of type Common_Node_Contents:
        with procedure save(file       : in text_io.file_type;
                            data       : in Common_Node_Contents);
        with procedure restore(file      : in      text_io.file_type;
                               data      : in out Common_Node_Contents);

        -- For saving and restoring a group of Common_Node_Contents objects that might be
        -- interconnected (in order to recover the interconnections):

        type CNC_Save_State is limited private;
        with procedure initialize(data : in out CNC_Save_State);
        with procedure finalize(data : in out CNC_Save_State);
        with procedure swap(left        : in out CNC_Save_State;
                            right       : in out CNC_Save_State);

        type CNC_Restore_State is limited private;
        with procedure initialize(data : out CNC_Restore_State);
        with procedure finalize(data : in out CNC_Restore_State);
        with procedure swap(left        : in out CNC_Restore_State;
                            right       : in out CNC_Restore_State);

        with procedure prepare_to_read_group(
                file                : in      text_io.file_type;
                read_state          : in out CNC_Restore_State);
                -- This routine is called before a group of objects of type
                -- Common_Node_Contents will be read from the specified FILE.
                -- This opportunity can be used to initialize the READ_STATE,
                -- and then load any information that was previously stored about
                -- the group as a whole into the into that state variable.
        with procedure read_one_of_a_group(
            file                : in      text_io.file_type;
            read_state          : in out CNC_Restore_State;
            data                : in out Common_Node_Contents);
            -- The DATA parameter of READ is mode in out because READ
            -- FINALIZEs the incoming value of DATA before placing the
            -- result of the READ operation in it.
        with procedure finish_reading_of_group(
            read_state : in out CNC_Restore_State);
            -- This routine is called once the reading of a group of objects
            -- of type Common_Node_Contents has been completed. This allows
            -- the READE_STATE to be cleaned up and finalized.

        with procedure prepare_to_write_group(
            write_state : in out CNC_Save_State);
            -- This routine is called before a group of objects of type
            -- Common_Node_Contents will be written to a file. It allows the
            -- WRITE_STATE, used for representing structural sharing
            -- information, to be initialized.
        with procedure write_marked_element(
            file                : in      text_io.file_type;
            write_state         : in out CNC_Save_State;
            data                : in      Common_Node_Contents);
            -- In effect, this routine sends a "copy" of DATA to FILE.
        with procedure finish_writing_group(
            write_state : in out CNC_Save_State);
            -- This routine is called once the writing of a group of objects
            -- of type Common_Node_Contents has been completed. This allows
            -- the WRITE_STATE to be cleaned up and finalized.

    package GPD_pkg is

        -- For use in single-threaded applications where concurrency
        -- protection for memory management is NOT necessary.

        type node_class is (gpd_empty,
                            gpd_integer,
                            gpd_boolean,
                            gpd_parent,
                            gpd_sequence,
                            gpd_user_defined);
            -- The type NODE_CLASS is left unencapsulated for simplicity

        type gpd_type is limited private;
            -- The basic operations for this type:
        procedure initialize(node : out gpd_type);
            -- ensures that NODE = NULL_GPD_NODE
            -- (NULL_GPD_NODE is the initial value for all elements of this
        -- type).
procedure finalize(data : in out gpd_type);
        -- If NODE is referenced from multiple locations, this reference
        -- is set to NULL_GPD_NODE. If this is the last remaining reference
        -- to NODE, then storage for NODE is reclaimed, and NODE is given
        -- the initial value, NULL_GPD_NODE.
procedure swap(left      : in out gpd_type;
               right     : in out gpd_type);
        -- exchanges the contents of left and right
procedure duplicate(original     : in     gpd_type;
                    copy         : in out gpd_type);
        -- DUPLICATE FINALIZE COPY first, then creates an identical
        -- copy of the ORIGINAL (including recursively copying all
        -- nodes pointed to by outgoing links--cycles are detected,
        -- and replicated appropriately).

null_gpd_node : constant gpd_type; -- the initial value for this type

----------------------------------------------------------------
-- The following 4 routines are common to all node classes.
-- They include functions to determine the class of a node,
-- to access the Common_Node_Contents slot of any node, and to save
-- or restore a GPD node to or from a file.
-- These functions are followed by 5 subpackages, one for
-- each node class. Each subpackage defines the node-class-specific
-- functions and procedures for a give node-class. Note that some
-- routines are overloaded (like "new_node", etc.) if the
-- desired node-class can be determined from the argument profile,
-- but ambiguous cases (like "new_node" for generating a
-- gpd_sequence vs. a gpd_empty) are given distinct names so
-- they do not have to be qualified with subpackage names.
--
function node_class_of(node : in gpd_type) return node_class;

-- The FREE routines in the previous versions are replaced by
-- the FINALIZE routine above.

procedure access_contents(node     : in out gpd_type;
                          data     : in out Common_Node_Contents);
        -- Exchanges the value (via a call to the appropriate SWAP)
        -- of DATA with the value of the Common_Node_Contents slot of
        -- NODE.

-- For saving and restoring a single GPD structure:                     |
procedure save(file      : in text_io.file_type;
               gpd       : in gpd_type);
procedure restore(file    : in      text_io.file_type;                  *
                gpd         : in out gpd_type);

-- For saving and restoring a group of GPD structures that might be
-- interconnccted (in order to recover the interconnections):

-- Saving:
--
-- Note that save/restore state objects cannot be copied or
-- compared, since those capabilities are not appropriate for
-- such state-holding objects.
--
type GPD_save_state is limited private;
procedure initialize(data : out GPD_save_state);
procedure finalize(data : in out GPD_save_state);
procedure swap(left         : in out GPD_save_state;
               right        : in out GPD_save_state);

type GPD_restore_state is limited private;
procedure initialize(data : out GPD_restore_state);
procedure finalize(data : in out GPD_restore_state);
procedure swap(left         : in out GPD_restore_state;
               right        : in out GPD_restore_state);

-- basic operations omitted for simplicity
procedure prepare_to_write_group(
        save_state : in out GPD_save_state);
procedure write_marked_element(
        file                : in     text_io.file_type;
        save_state          : in out GPD_save_state;
        data                : in     Common_Node_Contents);
procedure finish_writing_group(
        save_state : in out GPD_save_state);

-- Restoring:
procedure prepare_to_read_group(
        file                : in     text_io.file_type;
        read_state          : in out GPD_save_state);
procedure read_one_of_a_group(
        file                : in     text_io.file_type;
        read_state          : in out GPD_save_state;
        data                : in out gpd_type);
procedure finish_reading_of_group(
        read_state : in out GPD_save_state);


------------------ --------------------------------------------
-- This subpackage defines the functions available for gpd nodes
-- of class "gpd_empty." Each operation will ensure that its arg
-- is of class "gpd_empty," raising GPD_ERROR if otherwise.
--
-- A node of class "gpd_empty" has no outgoing nodes, and
-- no slots other than one to hold Common_Node_Contents.
--
package empty_node_pkg is
        procedure new_empty_node(node : in out gpd_type);
                -- Makes sure NODE is finalized, then replaces it
                -- with a node of class "gpd_empty", which can still
                -- hold Common_Node_Contents data.
end empty_node_pkg;



-- The routines in the following 2 packages use the SWAP
-- routines from the package BASIC_TYPE_SUPPORT to "access"
-- the internal contents of each node-class (gpd_integer and
-- gpd_boolean). This isn't strictly necessary for these
-- types (which may not be strongly encapsulated), but does
-- serve as an example of how strongly encapsulated types
-- would be treated.
--
-- In each case, an "access" routine replaces both the GET_DATA
-- and PUT_DATA routines found in other versions of this package.
-- If duplication of internal data were required by the user,
-- he could use the appropriate COPY or DUPLICATE routines
-- exported by BASIC_TYPE_SUPPORT (or whatever package defined
-- the type under consideration).

-----------------------------------------------------------------
-- This subpackage defines the functions available for gpd nodes
-- of class "gpd_integer." Each operation will ensure that its arg
-- is of class "gpd_integer," raising GPD_ERROR if otherwise.
--
package integer_node_pkg is
        procedure new_node(data      : in      integer;
                           node      : in out gpd_type);
        procedure access_data(node    : in out gpd_type;
                              data    : in out integer);
end integer_node_pkg;

-----------------------------------------------------------------
-- This subpackage defines the functions available for gpd nodes
-- of class "gpd_boolean." Each operation will ensure that its arg
-- is of class "gpd_boolean," raising GPD_ERROR if otherwise.
--
```

```
package boolean_node_pkg is
        procedure new_node(data       : in      boolean;
                           node       : in out gpd_type);
        procedure access_data(node    : in out gpd_type;
                              data    : in out boolean);
end boolean_node_pkg;


---------------------------------------------------------------
-- This subpackage defines the functions available for gpd nodes
-- of class "gpd_parent." Each operation will ensure that its arg
-- is of class "gpd_parent," raising GPD_ERROR if otherwise.
--
-- A node of class "gpd_parent" has an ordered list of children. The
-- length of this list is determined by the parameter to
-- Make_Empty_Parent_Node when the node was first created. The
-- children (some of which may be Null_GPD_Nodes) may be accessed
-- in any order by their relative positiions from the beginning of
-- the list.
--
package parent_node_pkg is
        procedure make_empty_parent_node(
                n          : in      positive := 2;
                node       : in out gpd_type);
        function max_children(node : in gpd_type) return natural;
                -- This routine remains a function, and relies on the
                -- COPY routine exported by BASIC_TYPE_SUPPORT. The user
                -- can't "access" the number of children--it must be read-only
                -- from his point of view.
        procedure access_child(
                child_node         : in out gpd_type;
                parent_node        : in out gpd_type;
                position           : in     positive);
                -- Remember, this _replaces_ the value of the Nth
                -- outgoing link with the value of CHILD_NODE, and
                -- replaces the value of CHILD_NODE with the value of
                -- the Nth outgoing link (a "swap").
        procedure put_child(
                child_node         : in     gpd_type;
                parent_node        : in out gpd_type;
                position           : in     positive);
                -- Rather than "swapping", this does what you would
                -- expect: it "links" the Nth outgoing link of the
                -- PARENT_NODE
                -- to the CHILD_NODE. This is a "restricted" form of
                -- aliasing, which is completely under the control of
                -- this module (i.e., not visible or accessible to the
          -- end user).
end parent_node_pkg;

----------------------------------------------------------------
-- This subpackage defines the functions available for gpd nodes
-- of class "gpd_sequence." Each operation will ensure that its arg
-- is of class "gpd_sequence," raising GPD_ERROR if otherwise.
--
-- A sequence node contains an arbitrarily long list of child
-- nodes, which may themselves be other sequences. These children
-- can be accessed, and the list of children modified, by the
-- subroutines in this package.
--
package sequence_node_pkg is
     subtype sequence_type is gpd_type;
          -- This subtype is just used for clarity in the
          -- declarations below to show where a node of class
          -- "gpd_sequence" is expected. If a node of a different
          -- class is used where this subtype appears, GPD_ERROR
          -- will be raised.
     procedure make_empty_sequence_node(node : in out gpd_type);
          -- Create a new gpd_sequence node with no outgoing links.


     -- Rather than "swapping", these operations introduces structural
     -- sharing semantics. They "link" the corresponding outgoing link
     -- of the SEQUENCE_NODE to the CHILD_NODE given in the
     -- argument (or remove such a link). This is a "restricted" form
     -- of aliasing, which is completely under the control of this
     -- module (i.e., not accessible to the end user).
     --
     procedure append(
          seq            : in out sequence_type;
          new_element    : in     gpd_type);
          -- adds a new outgoing link to the end of SEQ's
          -- list of links, then places a reference to the NEW_ELEMENT
          -- in this outgoing link.
     procedure remove_head(
          seq     : in out sequence_type;
          head    : in out gpd_type);
          -- removes the first outgoing link on SEQ's list, and
          -- returns the object pointed to by that link.
     procedure prepend(
          seq            : in out sequence_type;
          new_element    : in     gpd_type);
          -- like APPEND, but for the beginning of the list.
     procedure remove_tail(
          seq       : in out sequence_type;
          tail      :    out gpd_type);
          -- like REMOVE_HEAD, but for the end of the list.
     procedure access_nth_element(
          seq            : in out sequence_type;
          element        : in out gpd_type;
          N              : in     positive := 1);
          -- "Swaps" the node pointed to by the Nth outgoing link with
          -- the current value of ELEMENT.
     procedure consume(
          seq     : in out sequence_type;
          N       : in     positive := 1);
          -- Removes the Nth outgoing link from the list. FINALIZE is called
          -- on the contents before the link is removed.
     procedure consume_n_elements(
          seq     : in out sequence_type;
          N       : in     positive);
          -- Removes the first N outgoing links from the list. FINALIZE is
          -- called on the contents of each link before it is removed.
     function length(seq : in sequence_type) return natural;
          -- Returns the number of outgoing links.
     procedure reverse_sequence(seq : in out sequence_type);
          -- Reverses the order of the list of outgoing links.
     procedure copy(
          original          : in     sequence_type;
          duplicate         : in out sequence_type);
          -- Produces a new node of class "gpd_sequence" with an
          -- identical list of outgoing links. Unlike the DUPLICATE
          -- operation, however, both the ORIGINAL and the DUPLICATE conceptually
          -- share structural references to the same children (DUPLICATE
          -- would create a new set of identical children).
     procedure concat(
          onto      : in out sequence_type;
          from      : in out sequence_type);
          -- Removes all outgoing links from ONTO, concatenating them
          -- onto FROM's list of outgoing links. At completion,
          -- ONTO will have an empty list of links.
     function is_empty(seq : in sequence_type) return boolean;
          -- Are there any outgoing links from SEQ?

end sequence_node_pkg;

-----------------------------------------------------------------
-- This subpcckage defines the functions available for gpd nodes
-- of class "gpd_user_defined." This generic subpackage allows the
-- user to create GPD nodes that can contain any user-defined type.
-- Each operation will ensure that its arg is of class "gpd_user_defined,"
-- as well as ensuring that it contains data of the correct user-defined
-- type. The exception GPD_ERROR will be raised otherwise.
--
generic
    type user_defined_data is limited private;
    with procedure initialize(data : in out user_defined_data);
    with procedure finalize(data : in out user_defined_data);
    with procedure swap(left         : in out user_defined_data;
                        right        : in out user_defined_data);

    -- For saving and restoring a single object of type User_Defined_Data:
    with procedure save(file         : in text_io.file_type;
                        data         : in User_Defined_Data);
    with procedure restore(file      : in     text_io.file_type;
                           data      : in out User_Defined_Data);

-- For saving and restoring a group of User_Defined_Data objects that might be
-- interconnected (in order to recover the interconnections):

    type UDD_Save_State is limited private;
    with procedure initialize(data : out UDD_Save_State);
    with procedure finalize(data : in out UDD_Save_State);
    with procedure swap(left         : in out UDD_Save_State;
                        right        : in out UDD_Save_State);

    type UDD_Restore_State is limited private;
    with procedure initialize(data : out UDD_Restore_State);
    with procedure finalize(data : in out UDD_Restore_State);
    with procedure swap(left         : in out UDD_Restore_State;
                        right        : in out UDD_Restore_State);

    with procedure prepare_to_read_group(
            file             : in     text_io.file_type;
            read_state       : in out UDD_Restore_State);
            -- This routine is called before a group of objects of type
            -- User_Defined_Data will be read from the specified FILE.
            -- This opportunity can be used to initialize the READ_STATE,
            -- and then load any information that was previously stored about
            -- the group as a whole into the into that state variable.
    with procedure read_one_of_a_group(
            file             : in     text_io.file_type;
            read_state       : in out UDD_Restore_State;
            data             : in out user_defined_data);
            -- The DATA parameter of READ is mode in out because READ
            -- FINALIZEs the incoming value of DATA before placing the
            -- result of the READ operation in it.
    with procedure finish_reading_of_group(
            read_state : in out UDD_Restore_State);
            -- This routine is called once the reading of a group of objects
            -- of type User_Defined_Data has been completed. This allows
            -- the READE_STATE to be cleaned up and finalized.

        with procedure prepare_to_write_group(
            write_state : in out CNC_Save_State);
            -- This routine is called before a group of objects of type
            -- User_Defined_Data will be written to a file. It allows the
            -- WRITE_STATE, used for representing structural sharing
            -- information, to be initialized.
        with procedure write_marked_element(
            file            : in      text_io.file_type;
            write_state     : in out CNC_Save_State;
            data            : in      user_defined_data);
            -- In effect, this routine sends a "copy" of DATA to FILE.
        with procedure finish_writing_group(
            write_state : in out CNC_Save_State);
            -- This routine is called once the writing of a group of objects
            -- of type User_Defined_Data has been completed. This allows
            -- the WRITE_STATE to be cleaned up and finalized.

    package user_defined_node_pkg is

        procedure new_node(data      : in out user_defined_data;
                           node      : in out gpd_type);
        procedure access_data(node    : in out gpd_type;
                              data    : in out user_defined_data);

    end user_defined_node_pkg;

    ------------------------------------------------------------------
    -- Errors:
    -- This package only defines one exception, GPD_ERROR. This
    -- exception is raised whenever a node-class-specific function
    -- or procedure is called with an argument of the wrong class.
    -- The exception CONSTRAINT_ERROR is raised if NULL_GPD_NODE
    -- is passed into a routine.
    --
    gpd_error         : exception;

private
    type gpd_block(class          : node_class          := gpd_empty;
                   top_size       : natural             := 0;
                   bottom_size    : natural             := 0);
    type gpd_type is access gpd_block;
    null_gpd_node : constant gpd_type := null;
```

```
        type gpd_save_block;                                            |
        type gpd_save_state is access gpd_save_block;                   |
        null_gpd_save_state : constant gpd_save_state := null;          |

        type gpd_restore_block;                                         |
        type gpd_restore_state is access gpd_restore_block;             |
        null_gpd_restore_state : constant gpd_restore_state := null;    |

end GPD_pkg;
```

6. PROCEDURE VARIABLE *CONCEPT*

```
generic
    type arg_type is limited private;
    with procedure initialize(data : in out arg_type);
    with procedure finalize(data : in out arg_type);
    with procedure swap(left        : in out arg_type;
                        right       : in out arg_type);
package Procedure_Variable_Abstraction is

    type procedure_variable is limited private;
    procedure initialize(data : in out procedure_variable);
        -- This initializes a procedure variable to the conceptual
        -- value "NULL." This must be executed for each procedure_variable
        -- declared.
    procedure finalize(data : in out procedure_variable);
        -- This releases all resources associated with a procedure variable.
        -- It must be executed on each procedure_variable before that
        -- variable goes out of its defining scope.
    procedure swap(left        : in out procedure_variable;
                   right       : in out procedure_variable);

    function procedure_variable_is_null(pv : in procedure_variable)
        return boolean;
        -- Return TRUE iff PV has the conceptual value "NULL,"
        -- return FALSE otherwise.
    procedure reset_procedure_variable(pv : in out procedure_variable);
        -- Sets a procedure variable to the conceptual value "NULL."
        -- This routine is most often used to "erase" the value of
        -- a used procedure variable.

    generic
        with procedure P(a : in out arg_type);
        -- P should not access any variables outside itself (either
        -- global variables or variables in surrounding transient
        -- scopes).
    package Procedure_Definer is

        procedure set_procedure_variable_to_P(pv : in out procedure_variable);
            -- Sets PV to conceptually "point to" the procedure P.
```

```
          end Procedure_Definer;

          procedure invoke_procedure(pv    : in       procedure_variable;
                                     a     : in out arg_type);
                    -- If PV has the conceptual value "NULL," no action is taken.
                    -- If PV "points to" some procedure P, P is invoked with
                    -- A as its argument.

          UNINITIALIZED_PV : exception;
                    -- This exception is raised if a variable of type PROCEDURE_VARIABLE
                    -- is declared and then passed to an operation before INITIALIZE
                    -- has been called on it.

private
          type pv_block;
          type procedure_variable is access pv_block;
end Procedure_Variable_Abstraction;
```

7. PROCEDURE VARIABLE *CONTENT* USING TASKING

```
with unchecked_deallocation;
package body Procedure_Variable_Abstraction is

        task type go_between_type is
                entry in_args           (a      : in out arg_type);
                entry out_args          (a2     : in out arg_type);
                entry return_args(a3            : in out arg_type);
        end go_between_type;

        type procedure_type is access go_between_type;
        null_procedure : procedure_type := null;
        type pv_block is record
                pv : procedure_type := null_procedure;
        end record;

        procedure initialize(data : in out procedure_variable) is
        begin
                if data /= null then
                        finalize(data);
                end if;
                data := new pv_block'(PV => null_procedure);
        end initialize;

        procedure finalize(data : in out procedure_variable) is
                procedure free is new unchecked_deallocation(
                        pv_block, procedure_variable);
        begin
                if data /= null then
                        free(data); -- assigns data = null after deallocating space
                else
                        raise UNINITIALIZED_PV;
                end if;
        end finalize;

        procedure swap(left         : in out procedure_variable;
                       right        : in out procedure_variable) is
                temp : procedure_variable := left;
        begin
                -- The normal "swap" implementation. Note that it runs
        -- in "constant" time, regardless of the size of a
        -- PV_BLOCK, so the representation of a procedure variable
        -- can be altered without affecting its efficiency.
        left := right;
        right := temp;
end swap;

function procedure_variable_is_null(pv : in procedure_variable)
        return boolean is
begin
        if pv = null then
                raise UNINITIALIZED_PV;
        else
                return pv.pv = null_procedure;
        end if;
end procedure_variable_is_null;

procedure reset_procedure_variable(pv : in out procedure_variable) is
begin
        if pv = null then
                raise UNINITIALIZED_PV;
        else
                pv.pv := null_procedure;
        end if;
end reset_procedure_variable;

package body Procedure_Definer is

        task shell is
                entry receive_go_between(gb_holder : in procedure_type);
        end shell;

        go_between : procedure_type := new go_between_type;

        procedure set_procedure_variable_to_P(pv : in out procedure_variable) is
        begin
                if pv = null then
                        raise UNINITIALIZED_PV;
                end if;
                pv.pv := go_between;
        end set_procedure_variable_to_P;

        task body shell is
                gb    : procedure_type;
                a     : arg_type;
        begin
                initialize(a);  -- set it to a valid initial value
                accept receive_go_between(gb_holder : in procedure_type) do
                        gb:= gb_holder;
                end receive_go_between;
                loop
                        -- swap the requested argument value into A
                        gb.out_args(a);
                        -- invoke the actual procedure
                        p(a);
                        -- swap the (possibly altered) value back to the caller
                        gb.return_args(a);
                end loop;
                -- This point is unreachable, but for completeness,
                -- clean up when done :
                finalize(a);
        end shell;

begin
        shell.receive_go_between(go_between);
end Procedure_Definer;

procedure invoke_procedure(pv   : in       procedure_variable;
                           a    : in out arg_type) is
begin
        if pv = null then
                raise UNINITIALIZED_PV;
        elsif pv.pv /= null_procedure then
                pv.pv.in_args(a);
        end if;
end invoke_procedure;

task body go_between_type is
begin
        loop
                accept in_args(a : in out arg_type) do
                        -- accept input to procedue P
                        accept out_args(a2 : in out arg_type) do
                                -- put P's arg into a2 so SHELL task can see it
                                swap(a2, a);
                        end out_args;
                        accept return_args(a3 : in out arg_type) do
                                -- take output from SHELL task and put it back
                                -- in A to be passed back to INVOKE_PROCEDURE.
                                swap(a, a3);
                        end return_args;
                end in_args;
        end loop;
end go_between_type;
end Procedure_Variable_Abstraction;
```

8. PROCEDURE VARIABLE *CONTENT* USING *INTERFACE* PRAGMA

```
with system, unchecked_conversion, unchecked_deallocation;
package body Procedure_Variable_Abstraction is

        type arg_type_ptr is access arg_type;
        subtype procedure_type is system.address;
        null_proc : integer := 0;
        null_procedure : constant procedure_type := null_proc'address;
        type pv_block is record
                pv : procedure_type := null_procedure;
        end record;

        procedure initialize(data : in out procedure_variable) is
        begin
                if data /= null then
                        finalize(data);
                end if;
                data := new pv_block'(PV => null_procedure);
        end initialize;

        procedure finalize(data : in out procedure_variable) is
                procedure free is new unchecked_deallocation(
                        pv_block, procedure_variable);
        begin
                if data /= null then
                        free(data);  -- assigns data = null after deallocating space
                else
                        raise UNINITIALIZED_PV;
                end if;
        end finalize;

        procedure swap(left          : in out procedure_variable;
                       right         : in out procedure_variable) is
                temp : procedure_variable := left;
        begin
                -- The normal "swap" implementation.  Note that it runs
                -- in "constant" time, regardless of the size of a
                -- PV_BLOCK, so the representation of a procedure variable
                -- can be altered without affecting its efficiency.
        left := right;
        right := temp;
end swap;

function procedure_variable_is_null(pv : in procedure_variable)
        return boolean is
        use system;
begin
        if pv = null then
                raise UNINITIALIZED_PV;
        else
                return pv.pv = null_procedure;
        end if;
end procedure_variable_is_null;

procedure reset_procedure_variable(pv : in out procedure_variable) is
begin
        if pv = null then
                raise UNINITIALIZED_PV;
        else
                pv.pv := null_procedure;
        end if;
end reset_procedure_variable;

package body Procedure_Definer is

        procedure p_wrapper(aa : in system.address) is
                function from_sa is new unchecked_conversion(
                        system.address, arg_type_ptr);
                a : arg_type_ptr := from_sa(aa);
        begin
                p(a.all);
        end p_wrapper;

        procedure set_procedure_variable_to_P(pv : in out procedure_variable) is
        begin
                if pv = null then
                        raise UNINITIALIZED_PV;
                end if;
                pv.pv := p_wrapper'address;
        end set_procedure_variable_to_P;

end Procedure_Definer;

procedure invoke_procedure(pv   : in      procedure_variable;
                           a    : in out arg_type) is
        procedure c_invoke_hook(a   : in system.address;
                                   p   : in system.address);
            pragma interface(c, c_invoke_hook);
            use system;
        begin
            if pv = null then
                raise UNINITIALIZED_PV;
            elsif pv.pv /= null_procedure then
                c_invoke_hook(a'address, pv.pv);
            end if;
        end invoke_procedure;

    end Procedure_Variable_Abstraction;
```

9. C PROCEDURE VARIABLE INVOCATION USED BY *PROCEDURE_VARIABLE_ABSTRACTION* PACKAGE

```
void c_invoke_hook(a, p)
   int *a;
  void (*p)();
{
  (*p)(a);
}
```

10. PROCEDURE VARIABLE *CONCEPT* WITH SHARED "ENVIRONMENTS"

```
generic
        type arg_type is limited private;
        with procedure initialize(data : in out arg_type);
        with procedure finalize(data : in out arg_type);
        with procedure swap(left          : in out arg_type;
                            right         : in out arg_type);
package Procedure_Variable_with_Env_Abstraction is

        type procedure_variable is limited private;
        procedure initialize(data : in out procedure_variable);
                -- This initializes a procedure variable to the conceptual
                -- value "NULL." This must be executed for each procedure_variable
                -- declared.
        procedure finalize(data : in out procedure_variable);
                -- This releases all resources associated with a procedure variable.
                -- It must be executed on each procedure_variable before that
                -- variable goes out of its defining scope.
        procedure swap(left         : in out procedure_variable;
                       right        : in out procedure_variable);

        function procedure_variable_is_null(pv : in procedure_variable)
                return boolean;
                -- Return TRUE iff PV has the conceptual value "NULL,"
                -- return FALSE otherwise.
        procedure reset_procedure_variable(pv : in out procedure_variable);
                -- Sets a procedure variable to the conceptual value "NULL."
                -- This routine is most often used to "erase" the value of
                -- a used procedure variable.

        generic
                type environment_type is limited private;
                with procedure initialize(data : in out environment_type);
                with procedure finalize(data : in out environment_type);
                with procedure swap(left          : in out environment_type;
                                    right         : in out environment_type);

                with procedure p(a   : in out arg_type;
                                 e   : in out environment_type);
```

```
          package Procedure_Definer is

                    procedure set_procedure_variable_to_P(
                              pv    : in out procedure_variable;
                              e     : in out environment_type);
                              -- Sets PV to conceptually "point to" the procedure P.
                              -- Also takes the environment E and puts it in PV (upon
                              -- completion, E has the value given by "initialize").
                              -- Whenever PV is invoked, P will be called with E as
                              -- one of its parameters.

                    procedure access_procedure_environment(
                              pv    : in out procedure_variable;
                              e     : in out environment_type);
                              -- "Swaps" E with the environment stored in PV.

          end Procedure_Definer;

          procedure invoke_procedure(pv    : in      procedure_variable;
                                     a     : in out arg_type);
                    -- Invokes the procedure "P" which PV "points to," passing
                    -- it the argument A and the procedure environment stored
                    -- in PV.

private
          type pv_block;
          type procedure_variable is access pv_block;
end Procedure_Variable_with_Env_Abstraction;
```

11. UNIDIRECTIONAL ASSOCIATIVE MEMORY *CONCEPT* FROM SECTION 6.3.1

```
generic
        type Domain_Type is limited private; -- the domain of the associative map
        with procedure Swap(left, right : in out Domain_Type);
        with procedure Initialize(d : in out Domain_Type);
        with procedure Finalize(d : in out Domain_Type);
        with procedure Copy(from      : in      Domain_Type;
                            into      : in out Domain_Type);
        with function Is_Equal(left, right : in Domain_Type) return boolean;

        type Range_Type is limited private;  -- the Range of the associative map
        with procedure Swap(left, right : in out Range_Type);
        with procedure Initialize(r : in out Range_Type);
        with procedure Finalize(r : in out Range_Type);
        with procedure Copy(from      : in      Range_Type;
                            into      : in out Range_Type);
        with function Is_Equal(left, right : in Range_Type) return boolean;

package Unidirectional_Associative_Memory_Concept is

        Type UAM_map is limited private;
        -- basic operations defined for every type
        procedure Swap(left, right : in out UAM_map);
        procedure Initialize(m : in out UAM_map);
        procedure Finalize(m : in out UAM_map);
        procedure Copy(from      : in      UAM_map;
                       into      : in out UAM_map);
        function Is_Equal(left, right : in UAM_map) return boolean;

        -- primary operations for UAM_maps
        procedure get_default(m  : in      UAM_map;
                              r  : in out Range_Type);
                -- The value of R when the call is made is finalized; then a copy
                -- of the default value of M is placed in R.  M is not affected.

        function is_constant(m : in UAM_map) return boolean;
                -- Returns false if there exists a D in Domain_Type such that
                -- M(D) /= GET_DEFAULT(M).
```

```
    function is_not_default(m  : in UAM_map;
                            d  : in Domain_Type) return boolean;
        -- Returns true iff M(D) /= GET_DEFAULT(M) (i.e., if D has been
        -- entered into the map M).

    procedure reset(m  : in out UAM_map;
                    r  : in out Range_Type);
        -- finalizes all the old values in M, and resets it so that R is
        -- the new default value for M. The original default value for
        -- M is also finalized, and on exit R has been "consumed" and
        -- contains the value of a newly initialized Range_Type variable.

    procedure access(m  : in out UAM_map;
                     d  : in     Domain_Type;
                     r  : in out Range_Type);
        -- The map M is applied to the value D (D is used to "index" into
        -- M). The resulting Range_Type value is "swapped" with the value of
        -- R. On exit, R contains the old value of M(D), and M(D) contains
        -- the old value of R.

end Unidirectional_Associative_Memory_Concept;
```

[Peterson89a] [Suzuki82a] [Litvinchouk84a] [Goguen84b]

BIBLIOGRAPHY

DoD83a United States Department of Defense. 1983. *Reference Manual for the Ada Programming Language.* ANSI/MIL-STD-1815A-1983. Washington, D. C.: Government Printing Office.

NBS84a U.S. Dept. of Commerce, National Bureau of Standards. June 15, 1984. *Guideline on Software Maintenance.* FIPS PUB 106.

Abelson85a Abelson, Harold and Gerald Sussman. 1985. *Structure and Interpretation of Computer Programs.* Cambridge, MA: MIT press.

Aho86a Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers, Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley Publishing Co..

Biggerstaff87a Biggerstaff, Ted and Charles Richter. March, 1987. Reusability Framework, Assessment, and Directions. *IEEE Software* 4(no. 2): 41-49.

Bishop90a Bishop, Judy M. April, 1990. The Effect of Data Abstraction on Loop Programming Techniques. *IEEE Transactions on Software Engineering* 16(no. 4): 389-402.

Booch86a Booch, Grady. 1986. *Software Engineering with Ada, Second Edition.* Menlo Park, CA: Benjamin/Cummings Publishing Co..

Booch87a Booch, Grady. 1987. *Software Components with Ada.* Menlo Park, CA: Benjamin/Cummings Publishing Co..

Braun85a Braun, C. L., J. B. Goodenough, and R. S. Eanes. April, 1985. *Ada Reusability Guidelines.* Tech. Rept. 3285-2-208/2. Waltham, MA: SofTech, Inc..

Cohen89a Cohen, Norman H. September 7, 1989. *Ada Subtypes as Subclasses (Version 1).* RC 14912. T. J. Watson Research Center, Yorktown Heights, NY: IBM Research Division.

Cohen90a Cohen, Sholom. January 4-5, 1990. *Designing for Reuse: Is Ada Class-Conscious?.* Syracuse, NY: presentation at the "Realities of Reuse" workshop, CASE Center Software Engineering Workshop Series, Syracuse University.

Cox86a Cox, Brad. 1986. *Object-Oriented Programming: An Evolutionary Approach.* Reading, MA: Addison-Wesley.

Curtis89a Curtis, Bill. 1989. Cognitive Issues in Reusing Software Artifacts. *Software Reusability, Volume II: Applications and Experience*, ed. Ted J. Biggerstaff and Alan J. Perlis, 269-287. Reading, MA: Addison-Wesley Publishing Co..

Edwards89a Edwards, Stephen. July, 1989. *Toward More Reusable Ada Components.* accepted position paper for the "Reuse In Practice" workshop.

Edwards89b Edwards, Stephen. September, 1989. *A Conceptual Model for Reusable Components.* accepted position paper for the "Workshop on Language Issues for Reuse: Ada for the 90's.

Gargaro87a Gargaro, Anthony and T. L. Pappas. July, 1987. Reusability Issues and Ada. *IEEE Software*: 43-51.

Goguen83a Goguen, J. A., J. Meseguer, and D. Plaisted. 1983. Programming with Parameterized Abstract Objects in OBJ. *Theory and Practice of Software Technology*, ed. D. Ferrari, M. Bolognani, and J. Goguen, 163-193. Amsterdam, The Netherlands: North-Holland.

Goguen84b Goguen, J. A. 1984. *Suggestions for Using and Organizing Libraries for Ada Program Development.* Technical Report prepared for the Ada Joint Program Office. Menlo Park, CA: SRI International.

Goguen84a Goguen, J. A. September, 1984. Parameterized Programming. *IEEE Transactions on Software Engineering* SE-10(no. 5): 528-543.

Goguen86a Goguen, Joseph A. February, 1986. Reusing and Interconnecting Software Components. *IEEE Computer* 19(no. 2): 16-28.

Harms89a Harms, Douglas E. and Bruce W. Weide. March, 1989. *Efficient Initialization and Finalization of Data Structures: Why and How.* OSU-CISRC-3/89-TR11. Ohio State University.

Harms89b Harms, Douglas E. and Bruce W. Weide. March, 1989. *Types, Copying, and Swapping: Their Influences on the Design of Reusable Software Components.* OSU-CISRC-3/89-TR13. Ohio State University.

Krone88a Krone, J. August, 1988. *The Role of Verification in Software Reusability.* Columbus, OH: Ph.D. Dissertation, Dept. of Computer and Information Science, Ohio State University.

Krueger89a Krueger, Charles W. December 14, 1989. *Models of Reuse in Software Engineering.* CMU-CS-89-188. Pittsburgh, PA: Carnegie Mellon University.

Levy87a Levy, P. and K. Ripken. 1987. Experience in Constructing Ada Programs from Non-Trivial Reusable Modules. *Ada Components: Libraries and Tools - Proceedings of the Ada-Europe International Conference, Stockholm 26-28 May 1987*, ed. S. Tafvelin, 100 - 112.. Cambridge, U.K.: Cambridge University Press.

Liskov86a Liskov, B. and J. Guttag. 1986. *Abstraction and Specification in Program Development*. Cambridge, MA: MIT Press.

Litvinchouk84a Litvinchouk, S. D. and A. S. Matsumoto. September, 1984. Design of Ada Systems Yielding Reusable Components: An Approach Using Structured Algebraic Specification. *IEEE Transactions on Software Engineering* SE-10(no. 5): 544-551.

Luckham85a Luckham, D. and F. W. von Henke. March, 1985. An Overview of Anna, A Specification Language for Ada. *IEEE Software*: 9-22.

McNicholl86a McNicholl, D. G., et al. May, 1986. *Common Ada Missile Packages (CAMP)*. Tech. Rept. AFATL-TR-85-93. St. Louis: McDonnell Douglas Astronautics Co..

Mendal86a Mendal, G. O. 1986. Designing for Ada Reuse: A Case Study. *Proceedings Second International Conference on Ada Applications and Environments*, 33-42. Los Alamitos, CA: IEEE Computer Society Press.

Miller56a Miller, G. A. 1956. The Magical Number Seven Plus or Minus Two: Some Limits on Our Capacity to Process Information. *Psychological Review* 63: 81-97.

Muralidharan88a Muralidharan, S. and Bruce W. Weide. November, 1988. *On Distributing Programs Built from Reusable Software Components*. OSU-CISRC-11/88-TR36. Ohio State University.

Muralidharan89a Muralidharan, S. March 13-16, 1989. On Inclusion of the Private Part in Ada Package Specifications. *Proceedings of Seventh Annual National Conference on Ada Technology*. Atlantic City, NJ.

Musser87a Musser, D. R. and A. A. Stepanov. December, 1987. A Library of Generic Algorithms in Ada. *Proceedings of 1987 ACM SIGAda International Conference*. Boston.

Musser88a Musser, D. R. and A. A. Stepanov. July, 1988. Generic Programming. *Proceedings 1988 International Symposium on Symbolic and Algebraic*

*Computation*Lecture Notes in Computer Science. Springer-Verlag.

Musser89a — Musser, David R. and Alexander A. Stepanov. 1989. *The Ada Generic Library: Linear List Processing Packages.* New York, NY: Springer-Verlag.

Nielsen88a — Nielsen, Kjell and Ken Shumate. 1988. *Designing Large Real-Time Systems with Ada.* New York, NY: McGraw-Hill.

Noel86a — Noel, Robert W. October 5-9, 1986. DoD Standards: Hindrance or Help to Software Maintainability?. *MILCOM 86: 1986 IEEE Military Communications Conference*, 22.5/1-6. Monterey, CA: IEEE.

Parikh87a — Parikh, Girish. 47-48. It's a Dirty Job, But ... Someone's Got to Do Maintenance. *Computerworld.*

Parnas76a — Parnas, D. L. January, 1976. On the Design and Development of Software Families. *IEEE Transactions on Software Engineering* SE-2(no. 1): 1-9.

Peterson89a — Peterson, A. Spencer. June, 1989. *Coming to Terms with Terminology for Software Reuse.* Pittsburgh, PA: position paper for the Reuse in Practice Workshop, SEI.

Prieto-Diaz87a — Prieto-Diaz, Rubén and Peter Freeman. January, 1987. Classifying Software for Reusability. *IEEE Software* 4(no. 1): 6-17.

Shaw81a — Shaw, M. ed. 1981. *ALPHARD: Form and Content.* New York: Springer-Verlag.

Snyder87a — Snyder, A. 1987. Inheritance and the Development of Reusable Software Components. *Research Directions in Object-Oriented Programming*, ed. Bruce Shriver and Peter Wegner. MIT Press.

St. Dennis86a — St. Dennis, R. J., P. Stachour, E. Frankowski, and E. Onuegbe. March-April, 1986. Measurable Characteristics of Reusable Ada Software. *Ada Letters* 5(no. 2): 41-49.

Stevens79a — Stevens, W., G. Myers, and L. Constantine. 1979. Structured Design. *Classics in Software Engineering*, ed. Edward Yourdon, 207-232. New York, NY: Yourdon Press.

Suzuki82a — Suzuki, N. May 1982. Analysis of Pointer 'Rotation'. *Communications of the ACM* 25(no. 5): 330-335.

Tevanian87a — Tevanian, Jr., Avadis. December, 1987. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach*

Approach. CMU-CS-88-106. Pittsburgh, PA: Carnegie Mellon University.

Tracz87a Tracz, Will. 1987. Software Reuse: Motivators and Inhibitors. *Proceedings of COMPCON S'87*, 358-363. IEEE.

Tracz88a Tracz, Will. January, 1988. Software Reuse Myths. *Software Engineering Notes* 13(no. 1): 17-21. ACM SIGSOFT.

Tracz89a Tracz, Will. May/June, 1989. Parameterization: A Case Study. *Ada Letters* IX(no. 4): 92-102. SIGAda.

Tracz90a Tracz, Will. 1990. Implementation Working Group Summary. *Reuse in Practice Workshop Summary*, ed. James Baldo, Jr.. Alexandria, VA: to be published by the Institute for Defense Analyses.

Tracz90c Tracz, Will. 1990. *Formal Specification of Parameterized Programs in LILEANNA*. Stanford, CA: Ph.D. Dissertation, Dept. of Electrical Engineering, Stanford University.

Tracz90b Tracz, Will. January 4-5, 1990. *Where Does Reuse Start?*. abstract of presentation for the "Realities of Reuse" workshop, Syracuse University CASE Center.

Watt87a Watt, David A., Brian A. Wichmann, and William Findlay. 1987. *Ada: Language and Methodology*. Englewood Cliffs, NJ: Prentice Hall International.

Weide86a Weide, Bruce W. January, 1986. *Design and Specification of Abstract Data Types Using OWL*. OSU-CISRC-TR-86-1. Ohio State University.

Weide86b Weide, Bruce W. and A Catalog of OWL Conceptual Modules. January, 1986. OSU-CISRC-TR-86-2. Ohio State University.

Wileden88a Wileden, Jack C., Alexander L. Wolf, Charles D. Fisher, and Peri L. Tarr. 1988. PGRAPHITE: An Experiment in Persistent Typed Object Management. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston Massachusetts, November 28-30, 1988*, ed. Peter Henderson, 130-142. New York, NY: Association for Computing Machinery.

Distribution List for IDA Paper P-2378

NAME AND ADDRESS	NUMBER OF COPIES
Sponsor	
Lt Col James Sweeder SDIO/ENA The Pentagon, Room 1E149 Washington, DC 20301-7100	3
Others	
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
John Solomond AJPO OUSDRE/R&AT The Pentagon, Room 3E114 Washington, DC 20301-3081	1
Larry Latour Department of Computer Science University of Maine Neville Hall Orono, ME 04469-0122	1
Prof. Bruce W. Weide Dept. of Computer and Information Science The Ohio State University 2036 Neil Ave. Mall Columbus, OH 43210-1277	1
Will Tracz IBM Corporation Mail Drop 0210 Route 17C Owego, NY 13827-1298	1
S. Muralidharan 312 Knapp Hall Morgantown, WV 26506	1

NAME AND ADDRESS	NUMBER OF COPIES
George Mitchell MITRE MS Z676 7525 Colshire Drive McLean, VA 22102	1
Bill Frakes Software Productivity Consortium SPC Building 2214 Rock Hill Road Herndon, VA 22070	1
Doug Lea Visiting Researcher Syracuse University CASE Center 2-173 Science & Technology Center Syracuse, NY 13244	1
Robert Nelson Information Systems Management NASA Space Station Program Office 10701 Parkridge Blvd. Reston, VA 22091	1
Nelson Weiderman Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213-3890	1
Joanne Piper CIVPERS Stop H-3 Ft. Belvoir, VA 22060-5456	1
Christine Braun Contel Technology Center 15000 Conference Center Drive P.O. Box 10814 Chantilly, VA 22021-3808	1
Terri Payton UNISYS 12010 Sunrise Valley Reston, VA 22091	1

NAME AND ADDRESS	NUMBER OF COPIES
Jim Robinette DCA/Z4S/SMBA 3701 N. Fairfax Drive Arlington, VA 22203	1
Dr. John F. Kramer STARS Technology Center 1500 Wilson Blvd. Suite 317 Arlington, VA 22209	1
Joseph L. Linn Microsoft Corp. One Microsoft Way Redmond, WA 98052-6399	1
James Pennell 12569 Cavalier Drive Woodbridge, VA 22192	1
Jack Kleinert SAIC 1710 Goodridge Drive P.O. Box 1303 McLean, VA 22102	1
Mike Mitrione Dynamics Research Corp. 1755 Jefferson Davis Hwy, Suite 802 Arlington, VA 22202	1
Capt. Emily Andrew NTBJPO Falcon AFB, CO 80912	1
Ron Halbgewacks POET, Suite 300 1225 Jefferson Davis Hwy Arlington, VA 22202	1
Danny Holtzman Vanguard 10306 Eaton Place, Suite 450 Fairfax, VA 22030	1
Terry Starr GF POB 1000 Blue Bell, PA 19422	1

NAME AND ADDRESS	NUMBER OF COPIES
Jay Crawford Naval Weapons Center Code 31C China Lake, CA 93555	1
Harley Ham NAC-825 Naval Avionics Center 6000 East 21st Street Indianapolis, IN 46219-2189	1
Geree Streun General Dynamics/Ft. Worth Div. PO Box 748, MZ 4050 Ft. Worth, TX 76101	1
Ted Tenny General Dynamics/Ft. Worth Div. PO Box 748 MZ 1761 Ft. Worth, TX 76101	1
Ernie Roberts McDonnell Douglas Corp. PO Box 516 D309/B66/L2N/Room 210 St. Louis, MO 63166	1
Dan Edwards Boeing Military Airplanes PO Box 7730, K80-13 Wichita, KS 67277-7730	1
Capt. Tony Dominice Air Force ATF Liaison Army Aviation Systems Command AMCPEO-LHX-TM 4300 Goodfellow Blvd St. Louis, MO 63120-1798	1
Diane Foucher Naval Weapons Center Code 251 China Lake, CA 93555	1

NAME AND ADDRESS	NUMBER OF COPIES
LTC John Morrison Director, Interoperability NTBJPO Falcon AFB, CO 80912-5000	1
Capt. Jack Rothrock SofTech, Inc. 2000 North Beauregard Street Alexandria, VA 22311	1
Dennis Ahern Westinghouse Electronic Systems Group Aerospace Software Engineering, MS-432 PO Box 746 Baltimore, MD 21203-0746	1
Don Alley Software Productivity Consortium SPC Building 2214 Rock Hill Road Herndon, VA 22070	1
Richard Bremner G.E. Aerospace Suite 800 5933 W. Century Blvd Los Angeles, Ca 90045	1
John Gaffney Software Productivity Consortium SPC Bld 2214 Rock Hill Road Herndon, VA 22070	1
Todd Goodermuth GE Aerospace PO Box 1000 Blue Bell, PA 19422	1
Robert Holibaugh Software Engineering Institute Carnegie-Mellon Pittsburgh, PA 15213	1

NAME AND ADDRESS	NUMBER OF COPIES
Raj Kant Honeywell Systems and Research Center 3660 Technology Drive Minneapolis, MN 55418	1
Charles McNally Westinghouse Electronics Systems Group Contracts PO Box 746, MS-1112 Baltimore, MD 21203-0746	1
Bill Novak Software Engineering Institute Carnegie-Mellon Pittsburgh, PA 15213	1
Don Reifer Reifer Consultants, Inc. 2550 Hawthorne Blvd. Suite 208 Torrance, CA 90505	1
Alexander Allen Deputy Commander US Army SDC 106 Wynn Drive ATTN - SSAE-SD-GBR-S Huntsville, AL 35807-3801	1
Deane Bergstrom Chief, Software Engineering Branch Rome Laboratory/COEE Griffis AFB, NY 13441-5700	1
Gina Burt, Electronics Engineer WL/AARI-2 Wright Patterson AFB, OH 45433-6543	1
Sholom Cohen SEI Carnegie Mellon Software Engineering Institute 4500 5th Avenue Pittsburgh, PA 15213-3890	1

NAME AND ADDRESS	NUMBER OF COPIES
Craig Coleman DRC 1755 Jefferson Davis Hwy Suite 802 Crystal Square 5 Arlington, VA 22202	1
Tom Durek TRW MS FVA6/2050C 2701 Prosperity Avenue Fairfax, VA 22031	1
Bill Farrell DSD Labs 75 Union Avenue Sudbury, MA 01776	1
Marialena Finn SAIC 1710 Goodridge Drive MS T2-8-2 McLean, VA 22102	1
CPT Tim Fisk HQ SDIC Division/CN1 P.O. Box 92960 Los Angeles, CA 90009-2960	1
Harold Gann UIE 1500 Perimeter Parkway Suite 123 Huntsville, AL 35806	1
Terry Gill Carnegie Mellon University MS 8400 Falcon AFB, CO 80912-5000	1
Ron Green, Deputy Commander USA/SDC ATTN - SFAE-SD-GST-D 106 Wynn Drive Huntsville, AL 35807	1

NAME AND ADDRESS	NUMBER OF COPIES
Prof. James Hooper UAH Computer Science Bldg Room 111 Huntsville, AL 35899	1
Gary Mayes SDC/GSTS PO U.S. Army ATTN - SFAE-SD-TST 106 Wynn Drive Huntsville, AL 35807-3801	1
Malcolm Morrison UAH Computer Science Dept. Huntsville, AL 35899	1
Bill Novak GE/SEI Carnegie Mellon University Pittsburgh, PA 15213-3890	1
Frank Poslajko SDC MS CSSD-SP 206 Wynn Drive Huntsville, AL 35807	1
Ruben Prieto-Diaz SPC 2214 Rock Hill Road Clarendon, VA 22070	1
Rich Saik Teledyne Brown Engineering Cummings Research Park 300 Sparkman Drive MS-174 Huntsville, AL 35807-7007	1
Bob Saisi DSD Labs 75 Union Avenue Sudbury, MA 01776	1

NAME AND ADDRESS	NUMBER OF COPIES
Otis Vaugh	1
TBE	
Cummings Research Park	
300 Sparkman Drive	
MS 198	
Huntsville, AL 35807-7007	
Neal Winters	1
TBE	
Cummings Research Park	
300 Sparkman Drive	
MS 174	
Huntsville, AL 35807-7007	

CSED Review Panel

Dr. Dan Alpert, Director	1
Program in Science, Technology & Society	
University of Illinois	
Room 201	
912-1/2 West Illinois Street	
Urbana, Illinois 61801	
Dr. Thomas C. Brandt	1
10302 Bluet Terrace	
Upper Marlboro, MD 20772	
Dr. Ruth Davis	1
The Pymatuning Group, Inc.	
2000 N. 15th Street, Suite 707	
Arlington, VA 22201	
Dr. C.E. Hutchinson, Dean	1
Thayer School of Engineering	
Dartmouth College	
Hanover, NH 03755	
Mr. A.J. Jordano	1
Manager, Systems & Software	
Engineering Headquarters	
IBM Federal Systems Division	
6600 Rockledge Dr.	
Bethesda, MD 20817	

NAME AND ADDRESS	NUMBER OF COPIES
Dr. Ernest W. Kent Philips Laboratories 345 Scarborough Road Briarcliff Manor, NY 10510	1
Dr. John M. Palms, President Georgia State University President's Office University Plaza Atlanta, GA 30303	1
Mr. Keith Uncapher, Associate Dean School of Engineering University of Southern California Olin Hall 330A University Park Los Angeles, CA 90089-1454	1
IDA	
General Larry D. Welch, HQ	1
Mr. Philip L. Major, HQ	1
Dr. Robert E. Roberts, HQ	1
Ms. Ruth L. Greenstein, HQ	1
Dr. Richard J. Ivanetich, CSED	1
Ms. Anne Douville, CSED	1
Mr. Terry Mayfield, CSED	1
Ms. Sylvia Reynolds, manuscript, CSED	17
Dr. Richard L. Wexelblat, CSED	1
Beth Springsteen, CSED	1
David Hough, CSED	1
Norman Howes, CSED	1
Clyde Roby, CSED	5
Jon Wood, CSED	1
Cy Ardoin, CSED	1
Audrey Hook, CSED	1
Deborah Heystek, CSED	1
Robert Knapper, CSED	1
David Wheeler, CSED	10
Dennis Fife, CSED	10
James Baldo, CSED	1
Cathy McDonald, CSED	1
Reginald Meeson, CSED	1
David Carney, CSED	1
Richard Morton, CSED	1
John Boone, CSED	1
IDA Control & Distribution Vault	3